The ProMoT Manual

Martin Ginkel

Max-Planck-Institute for dynamics of complex technical systems,
Sandtorstr. 1, D-39106 Magdeburg, Germany
http://www.mpi-magdeburg.mpg.de/projects/promot/
E-mail: promot@mpi-magdeburg.mpg.de

19th August 2011

Abstract

ProMoT (**Pro**cess **Mo**deling **T**ool) is a modeling tool for the object-oriented and equation-based modeling of chemical, biochemical and intracellular processes for the simulation environments Diva and Diana. Mdl (**M**odel **D**efinition **L**anguage) is the object-oriented modeling language of ProMoT. The process unit models created with ProMoT contain algebraic and ordinary differential equations as well as specifications of process variables in Mdl notation. These models are written as CodeGenerator input files and can be numerically analyzed, simulated and optimized in the simulation environment Diva after calling the CodeGenerator. In ProMoT a process unit model is represented as a structural modeling entity that itself can be composed of structural and behavioral modeling entities. Structural modeling entities represent control volumes and signal transformers of a process unit model as well as their terminals. Behavioral modeling entities represent model equations and variables that may be attached to structural modeling entities. The modeling entities including the process unit models are formulated in Mdl. Mdl supports the definition of elementary modeling entities which can be aggregated to build composed modeling entities. By multiple inheritance subclasses of elementary and composite modeling entities may be created, where composite modeling entities inherit all parts of their superclasses. By aggregation and inheritance knowledge bases may be designed, whose modeling entities have standard interfaces as well as good documentations and may be reused and refined in a natural way.

This manual is an introductory guide to Mdl and the modeling environment ProMoT for experienced Diva and CodeGenerator users. The object model and language elements of Mdl are illustrated by considering simple modeling examples. The graphical user interface and the use of the ProMoT tool are also described.

Contents

1 Introduction **6**
1.1 ProMoT Architecture . . . 6
1.2 ProMoT models . . . 7
1.3 Introductory examples . . . 9
1.3.1 Mathematical model of the pure substance tank . . . 9
1.3.2 Model formulation of `tank` in Mdl . . . 11
1.4 Overview for the following chapters . . . 12

2 Object-oriented Modeling **13**
2.1 Aggregation . . . 13
2.2 Inheritance . . . 14
2.3 Goal driven Modeling . . . 14
2.4 Modeling entities . . . 14

3 The model definition language Mdl **16**

4 Structural Modeling Entities **19**
4.1 Modules . . . 19
4.1.1 Elementary Modules . . . 19
4.1.2 Composed Modules . . . 24
4.1.3 Indexed Modules . . . 27
4.1.4 Diva specific definitions . . . 27
4.2 Terminals . . . 29
4.2.1 Elementary Terminals . . . 29
4.2.2 Indexed Terminals . . . 30

5 Behavioral modeling entities **33**
5.1 Variables . . . 35
5.1.1 System-theoretic type . . . 35
5.1.2 Vectorial type . . . 36
5.1.3 Variable descriptions in Mdl . . . 37
5.2 Equations . . . 42
5.2.1 Mdl description . . . 42
5.3 Indexed variables and equations . . . 43

6 Relations between modeling entities **45**
6.1 Aggregations . . . 45
6.1.1 Strong and weak aggregations . . . 45
6.1.2 Internal part classes . . . 47

6.2 Coupling relations 48
6.3 Inheritance relations 48

7 The Graphical User Interface 51
7.1 ProMoT Browser 52
7.1.1 Browser Tree 53
7.1.2 Pull-down Menus 53
7.1.3 Pop-up Menu 56
7.2 Heritage Views 57
7.3 Aggregation View 57
7.4 ProMoT Visual Editor 58
7.5 ProMoT Visual Explorer 61
7.6 Text Editor 63
7.7 Model Output Dialog (for Diva) 64

A Mdl-Command reference 66
A.1 Strings and Symbols 66
A.2 Mathematic Expressions 66
A.3 Slots und Slot Paths 66
A.4 Names for modeling entities 66
A.5 Commands 67

B Introduction to EBNF 70

C Getting Started 72

D The basic elements of the library for logical modeling 76

E The basic elements of the library for dynamic modeling 82

F Shortcuts and modifiers of the ProMoT Visual Explorer 92

List of Figures

1 Architecture of ProMoT 7
2 Open ProMoT-model with three terminals. 8
3 Single phase liquid `tank` with inlet and outlet. 10
4 Specialization Hierarchy of the basic modeling entities. 14
5 Cascade of two tanks (`two-tanks`). 25
6 The flowchart of a tank with n_e inlets and n_a outlets called `mimo-tank` 30
7 Specialization hierarchy of system-theoretic types. 35
8 Flowchart of a single-phased multi-component tank with one inflow and one outflow. 43

9 Aggregation relations and internal part classes in the construction of `two-tanks` out of `tank` 47
10 Examples for multiple inheritance and inheritance priorization. 49
11 The ProMoT Browser 52
12 Pull-down menu of the browser. 53
13 Pop-up Menu of the browser. 56
14 Superclass view for the class `trans2a-mm_l`. 57
15 Aggregation view for `storage-intra_s`. 58
16 Visual editor for a cellular transporter module. 59
17 Recursive view of the module of the EGF signal transduction model. Different layers provide additional information, e.g. overview and navigation layer (on the bottom right). 62
18 Emacs for editing Mdl source code with error message. 63
19 The Model Output Dialog for writing a model to Diva. 64

1 Introduction

ProMoT (**Pr**ocess**Mo**deling**T**ool) is a knowledge-based tool for the equation-based, object-oriented modeling of chemical and biochemical processes. The process models constructed with ProMoT can be analyzed, simulated and optimized numerically in the simulation environment Diva after passing the CodeGenerator [Kö00]. Beginning in 2006 ProMoT is also capable to use the simulation environment Diana [KBM+06].

The object-oriented knowledge base of ProMoT contains process engineering knowledge represented by structural and behavioral modeling entities. Starting with these predefined modeling entities, new models can be constructed by object oriented modeling and then they can be added to the knowledge base.

Models in ProMoT are based on rigorous modeling with balance equations. Modules are the structural modeling entities that represent control volumes or signal transformers and therefore subsystems in the model. The model of the whole plant or apparatus that should be analyzed numerically is also represented as a module.

ProMoT provides two mechanisms to derive new modeling entities from existing ones:

- Aggregation
- (Multiple) Inheritance

Using aggregation on multiple levels the modeler can build new models by aggregating and connecting structural and behavioral modeling entities. With inheritance one can describe general aspects of a model in one modeling entity and then construct several special entities that inherit all the aspects from the general one.

Modeling entities in ProMoT are formulated in the object-oriented modeling language (**M**odel **D**efinition **L**anguage). This manual describes the object-oriented methodology, the semantics of the modeling entities in ProMoT and the language elements of Mdl. It will be explained with simple example models of tanks for pure and mixed liquids, how the mathematical model of an apparatus is described by structural and behavioral modeling entities.

1.1 ProMoT Architecture

ProMoT is, like the CodeGenerator, implemented in Common Lisp using the features of the Common Lisp Object System. As shown in Figure 1, ProMoT writes the simulation modules for Diva by using the CodeGenerator, which generates the Fortran source code from a non-structured symbolic model representation.

The ProMoT Gui (**G**raphical **U**ser **I**nterface) is implemented in Java[GJSB00] with the Java Foundation Classes (Swing) [tSd02] and runs as a separate process. The ProMoT Gui communicates with the ProMoT kernel via the Corba middle-ware. ProMoT can be started without the Gui as a command line compiler which is called `mdl2diva`. This is meant for fast generation of a Diva model within shell scripts.

Within the kernel modeling entities are represented in an internal data structure of specialized Clos class objects called Fdl. This classes can be created and manipulated using the Mdl interpreter or the visual editor of the Gui. Since both methods work directly on the same data, the modeler can switch between this two ways of modeling. The resulting models can be written from this internal representation into Mdl. The modeling language is therfore also used as storage format for the modeling libraries. The Gui incorporates the text editor Emacs for

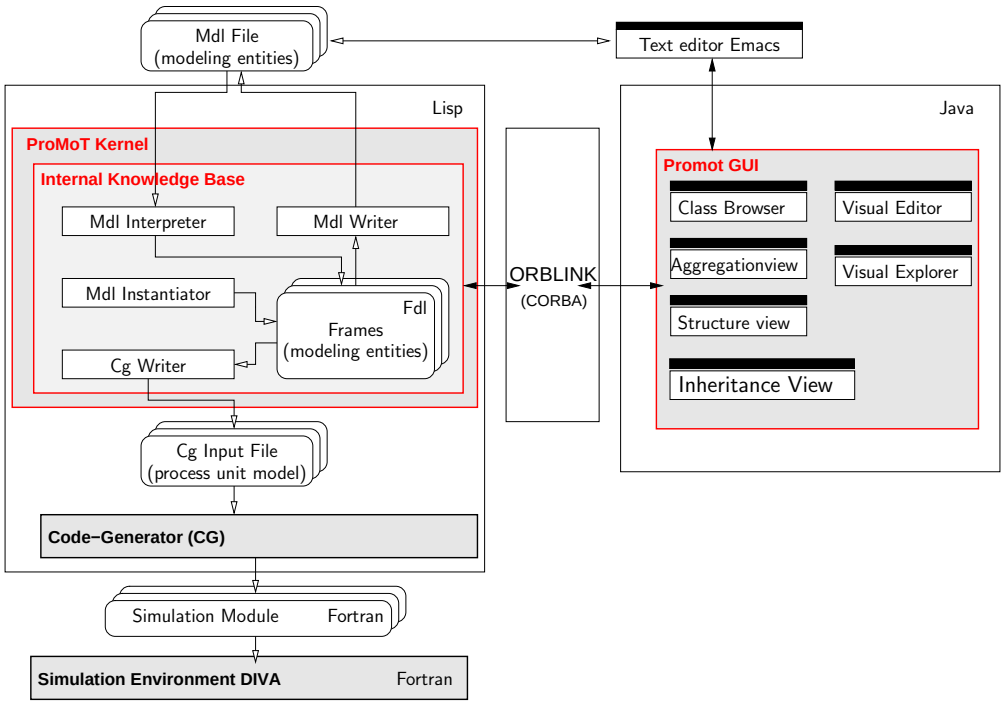


Figure 1: Architecture of ProMoT

changing the source code of the modeling entities. For installation instructions to use ProMoT see Appendix C. You can also try out the step by step tutorial found at `http://www.mpi-magdeburg.mpg.de/projects/promot/` under *Documentation*.

ProMoT makes it possible to use modeling knowledge bases. In the directory `/usr/local/promot/promot/kb` and its subdirectories some common knowledge bases for different modeling domains are stored. Further every modeler will have his personal set of modeling entities, that are used to form special models.

1.2 ProMoT models

This section describes the mathematical description of a Diva-model that can be built in ProMoT. ProMoT generates this model from the equation based and object-oriented form that is stored in Mdl. Although such a model normally contains the description of a whole system (closed model), it can also be connected to other Diva models (open model). A model is open if the module, describing the system in ProMoT and given out to Diva, has terminals. The variables within these terminals have to be directed inputs and outputs that can then be connected on one level within the Diva plant description file (.anl-File) to arbitrary Diva blocks (Created with the CodeGenerator or directly in Fortran). ProMoT models are elementary models within Diva. An example is given in Figure 2.

The system border of the model is described by the terminals that describe the exchange of material, momentum, energy and information with other models. Every terminal contains a set of input and output variables that are represented in the vectors $\mathbf{u}$ and $\mathbf{y}$ respectively (see also

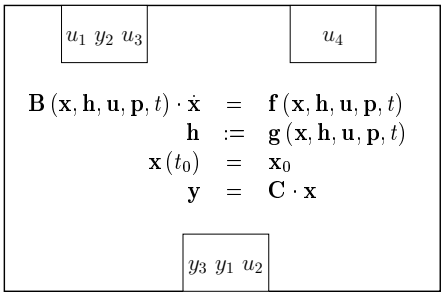


Figure 2: Open PROMOT-model with three terminals. Every terminal may have several input and output variables u_i and y_i, that are combined in this example to vectors $\mathbf{u} = (u_1\ u_2\ u_3\ u_4)^T$ and $\mathbf{y} = (y_1\ y_2\ y_3)^T$.

Figure 2). Within DIVA this variables are coupled with directed uni-directional (in the sense of data flow) connections. For more information about this, see [HKK+00].

A model generated by PROMOT contains the following vectors of variables:

- state variables $\mathbf{x} \in \mathcal{R}^{n_x}$;
- input variables $\mathbf{u} \in \mathcal{R}^{n_u}$;
- output variables $\mathbf{y} \in \mathcal{R}^{n_y}$;
- help variables $\mathbf{h} \in \mathcal{R}^{n_h}$;
- parameters that are constant in time $\mathbf{p} \in \mathcal{R}^{n_p}$.

The model is described by the following mathematical relations:

$$\begin{array}{lrcll} \text{DAE} & \mathbf{B}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t)\cdot\dot{\mathbf{x}} & = & \mathbf{f}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) & ;t > t_0 \\ \text{help variable assignment} & \mathbf{h} & := & \mathbf{g}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) & ;t > t_0 \\ \text{initial condition} & \mathbf{x}(t_0) & = & \mathbf{x}_0 & \\ \text{output selection} & \mathbf{y} & := & \mathbf{C}\cdot\mathbf{x} & ;t \geq t_0 \end{array} \tag{1}$$

The first line describes a system of differential and algebraic equations in a linear-implicit form. In the second line intermediate values are calculated by direct assignment,the so called help-variables, that can be used in the function $\mathbf{f}$ and in the matrix $\mathbf{B}$. Initial conditions for the DAE are given in the third line. For a DAE these conditions have to fulfill the system consistently. The last line defines the vector of outputs of the system as a part of the state vector. Every line of the matrix $\mathbf{C} \in \{0,1\}^{n_y \times n_x}$ contains $(n_x - 1)$ 0s and only one 1 as elements.

$$\mathbf{C} = \begin{matrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{matrix}$$

For the left hand side matrix $\mathbf{B}$, the right hand side vector function $\mathbf{f}$ and the vector of help functions $\mathbf{g}$ applies:

- $\mathbf{B} \in \mathcal{R}^{n \times n_x}$;
- $\mathbf{f} \in \mathcal{R}^n$ (n is the number of equations);

- $\mathbf{g} \in \mathcal{R}^{n_h}$.

The matrix $\mathbf{B}$ may be singular. The calculation of help variables is done via explicit directed assignments. This algebraic calculations of help variables may therefore not contain a dependency on the variable itself (i.e. they may not contain an algebraic loop). If this case arises, one of the help variables has to be transformed into a state variable and the assignment must be used as an algebraic equation.

For successful calculation of the model in DIVA the following conditions have to be fulfilled:

- The number of equations equals the number of states: $n = n_x$.
- The DAE in (1) may have a differential index of at most one.

Additionally the model in DIVA may contain time or state dependent event descriptions and switching schedules which represent discrete parts of the system. This topic will be expanded later.

1.3 Introductory examples

As introduction into the modeling language MDL the following examples are considered:

- a single phase liquid tank containing one pure substance (`tank`),
- a cascade of two tanks (`two-tanks`),
- a single phase multicomponent mixture tank (`mc-tank`).

In this section the mathematical model and the formulation in MDL of the `tank` is presented. The following sections of this manual describes the language elements that are used in this example. Furthermore it will be shown, how two instances of the `tank` are aggregated to form `two-tanks`, a cascade of two tanks.

Section 5 will present an alternative model formulation for the tank. In this version is a separate modeling-entity for every equation or variable defined and the model is extended to have an arbitrary number of inputs and outputs. The advantage of this kind of formulation is that the predefined variables and equations can be used in other models too. This is a first step towards building a standard knowledge base. But there is a tradeoff, because a compact definition of a modeling entity is replaced by a number of independent definitions that make the orientation in the model more difficult.

Later on in Section 5.3 a model for a tank containing a mixture of liquid substances (`mc-tank`) is derived from the simple `tank`. Therefore `mc-tank` is defined as a subclass of `tank`. The definition of `tank` is kept unchanged. The modeling entity `mc-tank` inherits all variables and equations of `tank`.

1.3.1 Mathematical model of the pure substance tank

The modeling task is to describe an ideally mixed tank with an inlet and an outlet by a mathematical model (Figure 3). The tank contains one substance that makes up exactly one thermodynamical phase. State variables of the system are:

- the molar holdup n of the liquid in the tank (dynamic state)

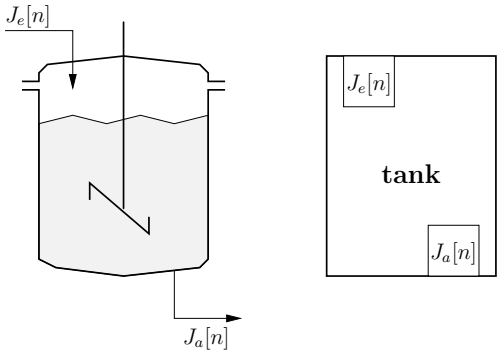


Figure 3: Single phase liquid tank with inlet and outlet. On the left hand side the process engineering flow sheet, on right hand side the graphical representation as modeling entity `tank` in ProMoT with two terminals.

- the molar flow out of the outlet (algebraic state, depending on the holdup)

There is only one input value, the molar flow $J_e[n]$ in the inlet, and one output in the outlet: $J_a[n]$, which is also a molar flow. In general the set of outputs is a subset of the state variables. Here the Vector of outputs is constructed with: $\mathbf{y} := \mathbf{C} \cdot \mathbf{x} = \begin{array}{cc} 0 & 1 \end{array} \cdot \mathbf{x} = \begin{array}{c} J_a[n] \end{array}^T$.
Model parameters are

- the gravity constant of earth $g = 9.81 m/s^2$;
- the molar concentration c of the liquid;
- the average cross section A of the tank;
- the cross section A_a of the outlet.

The model equations are formed in a balance view on the modeling task. The flow-sheet of the system in Figure 3 is a control volume that contains the molar holdup n of the liquid. The change of n in time is equal to the sum of all flows across the system border:

$$\frac{dn}{dt} = J_e[n] - J_a[n] \qquad ; t > t_0 \ . \tag{2}$$

The initial condition of the dynamic state n is the holdup n_0 of the tank in time t_0.

$$n(t_0) = n_0 \ . \tag{3}$$

The last equation calculates the algebraic state variable $J_a[n]$. The kinetic energy of the liquid inside the tank should be neglected. Therefore the flow out of the outlet is determined by Torricellis law:

$$J_a[n] = cA_a\sqrt{2gh} \quad \text{combined with the liquid level} \tag{4}$$

$$h = \frac{n}{cA} \tag{5}$$

$$J_a[n] = A_a\sqrt{2\frac{gcn}{A}} \tag{6}$$

The whole equation system for the model consists of the differential equation (2) and the algebraic equation (6):

$$\begin{array}{rcll} \frac{dn}{dt} & = & J_e[n] - J_a[n] & ; t > t_0 \\ J_a[n] & = & A_a\sqrt{2\frac{gcn}{A}} & ; t \geq t_0 \\ n(t_0) & = & n_0 & \end{array} \tag{7}$$

1.3.2 Model formulation of `tank` in Mdl

The following Mdl command defines the model of a liquid tank with one phase as module `tank`.

```
(define-module :class "tank"
  :super-classes ("module")
  :documentation "Pure substance tank"
  :properties (("aggregate_state" :one-of ("liquid" "gas")
                                  :value "liquid")
               ("spatial_resolution" :one-of ("distributed" "concentrated")
                                     :value "concentrated"))
  :terminals (("in" :is-a "terminal"
                    :variables (("jn" :is-eq-to "parent.jen")))
              ("out" :is-a "terminal"
                     :variables (("jn" :is-eq-to "parent.jan"))))
  :variables (("n" :is-a "variable"
                   :system-theoretic "state"
                   :value "100.0d0"
                   :minimum "0.0d0"
                   :maximum "200.0d0"
                   :unit "kmol"
                   :report "all"
                   :documentation "molar holdup")
              ("jen" :is-a "variable"
                     :system-theoretic "input"
                     :value "10.0d0"
                     :unit "kmol/h"
                     :documentation "molarer inflow")
              ("jan" :is-a "variable"
                     :system-theoretic "output"
                     :value "10.0d0"
                     :unit "kmol/h"
                     :report "all"
                     :documentation "molarer outflow")
              ("a" :is-a "variable"
                   :system-theoretic "real-parameter"
                   :value "1.0d0"
                   :unit "m^3"
                   :documentation "cross section of tank")
              ("aa" :is-a "variable"
                    :system-theoretic "real-parameter"
                    :value "0.1d0"
                    :unit "m^3"
                    :documentation "cross section of outlet")
              ("c" :is-a "variable"
                   :system-theoretic "real-parameter"
```

```
                    :value "55.0d0"
                    :unit "kmol/m^3"
                    :documentation "molar concentration of water")
               ("g" :is-a "variable"
                    :system-theoretic "real-parameter"
                    :value "9.81d0"
                    :unit "m/s^2"
                    :documentation "gravity constant"))
    :equations (("bal_n" :is-a "equation"
                         :relation ":diff<t> n == jen - jan"
                         :documentation "total material balance")
                ("fd_out" :is-a "equation"
                          :relation "0.0d0 == jan*jan -
                                              aa*aa * 2.0d0 * g * c * n/a"
                          :documentation "torricellis law for the outlet")))
```

1.4 Overview for the following chapters

Section 2 gives a short introduction to the basic principles of object-oriented modeling and the basic modeling entities that are available in ProMoT. In Section 3 the model definition language Mdl is described. The following sections 4 and 5 explain the properties of the structural and behavioral modeling entities in detail and illustrate how to define these with Mdl. Section 6 discusses the aggregation and inheritance relations between the different modeling entities. Section 7 introduces the different parts of the ProMoT Gui and their function. It also explains how the user can model with the ProMoT Visual Editor and the ProMoT Visual Explorer. A full command reference for Mdl can be found in Appendix A. For syntax descriptions in the whole manual EBNF is used which is introduced shortly in Appendix B and Appendix C presents instructions, how to install and start ProMoT successfully.

2 Object-oriented Modeling

Models in the form of (1) can be built within ProMoT using object-oriented modeling. Therefore the considered system has to be isolated from its environment. The model of the system itself will then be aggregated from structural and behavioral modeling entities.

Structural modeling entities (SME) describe a segmentation of the system under consideration. In process engineering the spatial structure is often used as the basis of the structure. For a good structural design of the model in ProMoT it is recommended to use the network theory [Gil97]. There are two different types of SME: modules and terminals.

Modules can be storage-containing components or storage-free connection-elements (see [Gil97] for details). Most Models can be structured into components and connection-elements. Modules are "separatable material parts" ([Mar96], page 46) of the system under consideration, e g. control volumes (phases, phase borders, walls), control surfaces (phase border surfaces), signal-transformers (controllers), storages (substances in a phase) and transformers (reactions in a phase). The border of a module is described by its terminals. With the terminals the module can exchange material, momentum, energy and information. The model of the complete system is also a module.

The mathematical model of a module consists of associated behavioral modeling entities (BME). BME are scalar or indexed ordinary differential equations, algebraic equations, state variables, inputs, outputs, parameters, intermediate variables and indices. With the association of BME to a module and its terminals a mathematic model is formed that is similar to (1).

A set of modules can be aggregated to a more complex module. Therefore the terminals of the (sub-)modules are connected or propagated as terminals of the complex module. This aggregation-step can be repeated to built up a hierarchy of modules up to the module of the complete system. The mathematical model of an aggregated module is formed by its submodules and coupling equations for all connected terminals.

- Aggregation,
- (Multiple-)Inheritance.

2.1 Aggregation

By aggregating a model from structural and behavioral modeling entities an aggregation hierarchy is built. In this hierarchy modeling entities are related by aggregation and coupling. A modeling entity can consist of other modeling entities. The compound modeling entity is called a container and the aggregated entities are called parts. Encapsulation of modules is an important quality of object-oriented modeling. Without terminals a module is completely separated from its environment. From the outside its state and behavior are invisible. Only through the terminals a selected subset of its variables can be accessed[1].

In the example Section 1.3.2 all terminals, equations and variables are parts of the container `tank`. The terminal `in` has as part the flow `jen` of the tank, the terminal `out` contains the flow `jan`. Obviously a modeling entity can occur as part of different containers. In the `tank` `jan` is a strong part of `tank` and a weak part of the terminal `out`. More details about aggregation and coupling relations, containers, strong and weak parts are to be found in Section 6.

[1] This is the "ideal" modeling paradigm. By using slot-paths (see Section 6.1) it is possible to break the encapsulation. This is sometimes helpful to express some special model structure.

2.2 Inheritance

In ProMoT a specialization hierarchy or ontology exists which expresses the inheritance relations between the modeling entities. A modeling entity can be defined as subclass of one or several other modeling entities (super-classes). Between sub- and superclasses exist inheritance relations. Therefore a subclass inherits all parts and attributes of its superclasses. A subclass can add new parts or attributes, or it can override or specialize inherited attributes. A subclass may not omit some inherited part[2].

2.3 Goal driven Modeling

While aggregating a simulation model in an object-oriented fashion, a good design of the modules is important. The modeling is based on balance equations and goal driven and shall produce a model that matches the requirements of the modeling task. Therefore the granularity of the modules and model equations have to be chosen accordingly. The choice of the module's border is arbitrary, but it should take the physical properties of the system and its spatial structure into account.

When different modules are coupled to a systems model the modeler has to take care that the DAE (1) has a differential index less or equal than one. Most numerical simulation algorithms (integrators) are only capable to handle systems that fulfill this condition. An index greater than one occurs, if dynamic state variables[3] in a link between two modules are set equal. A physical explanation for this is that the storages in the two models are set equal and are therefore not distinguishable. The problem can be solved if the model structure is changed, so that only one common storage is present in this system and is balanced for the whole system. A better solution is a general design according to the network theory, where components (i.e. storages) are never directly coupled, but have connection elements between them.

2.4 Modeling entities

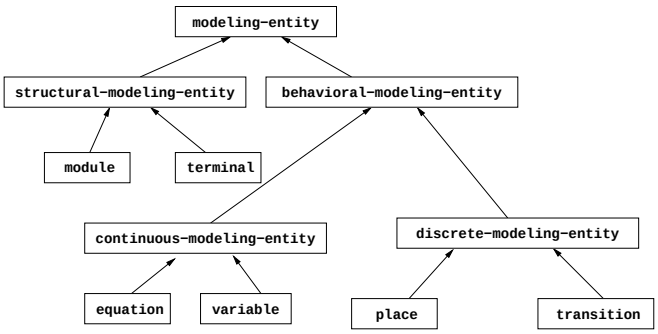


Figure 4: Specialization hierarchy of the basic modeling entities in ProMoT. The arrows denote inheritance relationships.

Since version 0.6.2, ProMoT provides the basic modeling entities depicted in Figure 4. Starting with these entities further modeling entities can be defined by inheritance and aggregation. New

[2]This is necessary because other parts inherited from the superclass will become inconsistent if they rely on the omitted attribute.

[3]Dynamic states x are those that appear in the derivative expression $\frac{dx}{dt}$ in a differential equation.

modeling entities are in any case subclasses of the basic modeling entities `module`, `terminal`, `variable`, `equation`, `place`, and `transition`. ProMoT provides the feature of multiple inheritance but the subclasses of the basic modeling entities must form distinct class hierarchies, i. e. no modeling entity may have two of the basic classes as direct or indirect superclass. Basic modeling entities are present in ProMoT after the start of the program. The user can also load his or her favorite standard classes from a knowledge base by adding some commands `(include <filename>)` into his or her `${HOME}/.promotrc` .

3 The model definition language Mdl

Modeling entities can be created using the object-oriented modeling language MDL . In MDL the following commands can be used:

- `include`
- `cd`
- `define-module`,
- `define-terminal`
- `define-variable`
- `define-equation`

The first two commands are used to work with structured source files and directories. `include` loads a MDL source file and `cd` changes the current directory of the PROMOT system. All files that are `included` after this point without fully specified path, are taken from this directory. This directory is also the default directory for all writing actions. All other commands define new modeling entities and are therefore named definition commands. The modeling entities are integrated into the specialization hierarchy as subclasses of the basic modeling entities shown in Figure 4.

The syntax of these commands follows the rules of LISP, every syntactic group is enclosed in parenthesis and all options of a command start with a colon (e. g. `:class`) followed by a value. With one definition command exactly one modeling entity is created. All definition commands have the following common options[4]:

`:class` ⟨**class-name**⟩ [**non-optional**]

The (unique) name for this modeling entity.[5] This name identifies the modeling entity. If a definition command defines a modeling entity with an already existing name, the old definition is overwritten. Excerpt from example in Section 1.3.2:

```
1          (define-module :class "tank"
```

`:super-classes` ⟨**list-of-class-names**⟩ [**non-optional**]

Definition of the superclasses of a modeling entity.

The new modeling entity has an inheritance relationship with these classes (see Section 6.3). Excerpt from example in Section 1.3.2:

```
2          :super-classes ("module")
```

`:documentation` ⟨**quoted-string**⟩ [**optional**]

Textual informal description of the modeling entity.

This is used as documentation in the generated output files or as information in the graphical user interface. Excerpt from example in Section 1.3.2:

[4] The order of all command and attribute options inside MDL commands is arbitrary.

[5] The syntax of the MDL commands and options is given in the Extended Backus Naur Form (EBNF) Notation. Appendix B introduces EBNF. The complete definition of all MDL commands is given in Appendix A.

```
3    :documentation
4          "Pure substance tank."
5
```

`:properties` `(` {⟨**property-slot**⟩}$^+$`)` [**optional**]

Definition of a list of property attributes , that describe the physical or chemical phenomena and modeling assumptions formally. Excerpt from example in Section 1.3.2:

```
4    :properties (("aggregate_state" :one-of ("liquid" "gas")
5                                    :value "liquid")
6                 ("spatial_resolution" :one-of ("distributed" "concentrated")
7                                       :value "concentrated"))
```

{⟨properties-slot⟩}$^+$ means that a list of property attributes must contain at least one element (see Appendix B). Generally a definition of a property attribute looks like this:

⟨property-slot⟩ :== `(` ⟨slot-name⟩ `:value` ⟨quoted-string⟩
[`:one-of` ⟨list-of-quoted-strings⟩]
`)`

`:indices` `({`⟨**index-definition**⟩`}`$^+$`)` [**optional**]

Definition of a list of indices.

If this option is specified, an indexed (array) modeling entity is created. It is possible to define multiple indices, therefore every index-dimension corresponds to one entry in the list of indices `({`⟨index-definition⟩`}`$^+$`)`. In general an index definition looks like:

⟨index-definition⟩ :==
`(:index` ⟨slot-name⟩ `:lower` ⟨math-expression⟩ `:upper` ⟨math-expression⟩ `)`

With the keyword `:index` the run variable is given, `:lower` and `:upper` specify the lower and upper border of the indexed interval. There are multiple examples of indices in the following example.

`:model-name` ⟨**quoted-string**⟩ [**optional**]

The `:model-name` specifies a short name that should be used for this modeling entity (and possible strong parts of it) within Diva .

Since names in Diva are limited to relatively short fixed string lengths, ProMoT can not write out a model with the full hierarchical names for all variables, but it shortens these names to the allowed limits. The automatically chosen alias names may lead to confusion during simulation. To fix the names of important variables to meaningful values for the access during simulation, one can specify a fixed alias name by `:model-name`. This name should be unique in the whole model that is written to Diva. Otherwise ProMoT changes the names to unique names by appending numbers at the end. Modules define the first part (2-4 characters) of the name of all directly contained variables. To this prefix usually the full variable name inside the module is appended (if this is not too long). If the name of the module is fixed by `:model-name`, the names of all variables it strongly contains, get changed in one step.

Additionally to the above mentioned, every definition command may have other options. Amongst others there are options that define parts (see Section 6.1). Parts are named attributes whose values are modeling entities itself. If a modeling entity has parts, it is called a container. The names ⟨slot-name⟩ of parts and attributes are alphanumerical character strings, that must start with a letter and may contain the under bar „_“ as the only special character. Every name of a part or attribute must be unique in the context of its container. The name `t` is reserved for the simulation time and can not be used for user defined parts. In the following sections the commands `define-module`, `define-terminal`, `define-variable` and `define-equation` are discussed in detail. When ProMoT reads this Mdl commands, it checks their syntax and translates them into Fdl commands.

Fdl (**F**rame **D**efinition **L**anguage [Trä98]) provides the data structures for the internal storage of ProMoT based on a knowledge representation, so called frames. Fdl is specialized in representing containers as aggregations of parts and specializations of superclasses. By executing the Fdl-commands the modeling-entities defined in Mdl are translated into class objects that are the physical storage for a frame of Fdl. This class objects can be directly manipulated using the ProMoT Visual Editor (see Section 7). They are also able to generate their own Mdl command to store their definition in a file.

4 Structural Modeling Entities

The modeling entity `structural-modeling-entity` is the ancestor of all structural modeling entities, and therefore of all modules and terminals (see also Figure 4). This section is divided in two parts. The first part describes the properties of modules and explains, how modules with the command `define-module` can be defined as subclasses of `module`. Furthermore at the example of a tank cascade is shown, how elementary modules can be aggregated to form compound modules. The second part deals with the definition of terminals as subclasses of `terminal`.

4.1 Modules

Modules are "separatable material parts" ([Mar96], page 43) of the modeled system. Examples are control volumes (phases, phase borders and walls), areas (surfaces of phases) and signal transformers (controllers). The systems border of a module is described by its terminals. Through a terminal the module can exchange material, momentum, energy and information with adjacent modules. `Module` is a direct subclass of the `structural-modeling-entity`. A new module is defined by the command `define-module`. The elements of the list `:super-classes` (Section 3) must be either direct or indirect subclasses of the class `module` or the class `module` itself. Superclasses of another basic type (e.g. `terminal`) are not allowed and cause an error. In the simplest case (see example in Section 1.3.2) a module is a direct subclass of `module`.

In addition to the universal command options of Section 3 (`:class`, `:super-classes`, `:documentation`), `define-module` has the options `:terminals`, `:variables`, `:equations`, `:places`, `:transitions` and `:modules` which attach parts to this module. Differently spoken the module is defined as composed modeling entity, as a container that contains these parts. Three types of modules can be distinguished:

elementary modules are modules that contain no other modules as parts of them. They only contain variables and equations.

composed modules are composed out of other modules that they contain.

indexed modules are composed modules that form arrays of other modules. Examples therefore are cascades, distillation-column-sections and discretized distributed systems.

4.1.1 Elementary Modules

Encoding in Mdl: Elementary modules contain only the command options `:terminals`, `:variables`, `:equations`, `:places` and `:transitions`. All those options define lists of named parts. A named part is identified by a unique name throughout the module[6] and it contains an instance of the requested type (e. g. `terminal`, `variable` ...).

`:terminals` `(` {⟨**terminal-slot**⟩}$^+$ `)` [**optional**]

Definition of the terminals of a module.

Excerpt from the example in Section 1.3.2:

```
8         :terminals (("in" :is-a "terminal"
9         :variables (("jn" :is-eq-to "parent.jen")))
```

[6] One module defines one separated name space. One name may be only used once to identify a part in the union of the sets of `:terminals`, `:equations` and so forth.

```
10          ("out" :variables (("jn" :is-eq-to "parent.jan"))))
11
```

The terminals of the module `tank` are defined as `in` and `out`. Both attributes get in this case internal subclasses of the class `terminal` which are automatically generated by ProMoT.[7] This internal classes contain some weak parts that are assigned with the command option `:variables`. The terminal `in` contains the part with the name `jn` which is a reference to the part `jen` of the module `tank`, which is in this case the "`parent`" of `in`. The terminal `out` gets the weak part `jn` respectively which refers to the part `jan` of the `tank`. Therefore `in` contains the input variable `jen` of the module tank whereas `out` contains the output variable `jan`. But in both cases the different variables are named `jn` inside the terminal (`in` or `out`). This assignment exports the variables as an interface to the module `tank`.

The example `tank` is a device model. As described in Section 1.2 the terminal variables of a device model are written as input or output variables to the regarding CodeGenerator input file. In this case the variables are `jen` and `jan`.

$\{\langle$terminal-slot$\rangle\}^+$ allows a list of terminal definitions which must contain at least one terminal (see also Appendix B). In general the definition of a terminal can use the following options:

⟨terminal-slot⟩ :== `(` ⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties (` $\{\langle$property-slot$\rangle\}^+$ `)`]
[`:indices (` $\{\langle$index-definition$\rangle\}^+$ `)`]
[`:variables (` $\{\langle$weak-slot$\rangle\}^+$ `)`] | ⟨weak-slot⟩

A terminal can not only be a strong slot with the command option `:is-a` (or no `:is-xxx` option at all) but it can also be a weak part of the module. In this case it is a reference to a terminal of an internal submodule. This can only appear in compound modules (see below). In this place only strong parts are considered. The part options `:documentation`, `:properties` and `:indices` correspond to the command options of the command `define-terminal`, which is explained in more detail in Section 4.2.

The part options `:is-a` and `:variables` differ from the command options of `define-terminal`.

`:is-a` ⟨**class-name**⟩ **[optional]**

Definition of the class which should be instantiated for the defined terminal part. The type ⟨class-name⟩ must be an arbitrary subclass of `terminal` or `terminal` itself. If `:is-a` is not given by the user then (if defined) the class of an equally-named inherited part of a superclass is used or it defaults to the class `terminal`. In this example the definition uses the class `terminal` in both cases: for `in` this is specified explicitly, in the case of `out` it defaults implicitly to that class. As mentioned above, ProMoT creates internal subclasses of `terminal` for each strong part that is created and uses an instance of this subclass as part of the defined module. By the creation of these subclasses the defined terminal can have other and more local part options (e. g. `:variables` and `:documentation`) as the class referred to by `:is-a` (see also Section 6).

[7] Parts and internal subclasses are described in Section 6 in more detail.

`:variables` ({⟨**weak-slot**⟩}$^+$) **[optional]**

Assignment of variables to this terminal.

This definition looks like the following:

⟨weak-slot⟩ :== (⟨slot-name⟩ `:is-eq-to` ⟨slot-path⟩)

⟨slot-name⟩ is the name that this variable gets inside the terminal. This is important for the connection of terminals with links, because the variables are linked according to their names inside the terminal and regardless to their original names inside the module. The general form of the part option for the definition of weak part is:

`:is-eq-to` ⟨**slot-path**⟩ **[non-optional]**

Definition of a so called slot-path that is referred by the just defined part. This slot-path leads to another part in the model. It is a composed hierarchical name where the single parts are separated by a dot. Every name refers to a part in the model. The name "`parent`" is a special implicit name which refers to the (*one*) strong container of the part. The reference is created to the last part of the path (see also Section 6.1). In this example

```
8            ("in" :is-a "terminal"
9            :variables (("jn" :is-eq-to "parent.jen")))
10
```

`jn` denotes a reference to the part `jen` of the strong container `parent` of the terminal `in`. This is the module `tank`, which possesses `jen`. The ⟨slot-path⟩ must refer to a variable out of the following system-theoretic types (see Section 5.1): `"state"`, `"input"`, `"output"`, `"structure-parameter"`, `"index"`. If the model should be used as device model that is connected with other device models in a DIVA `.anl` file then variables of the type `"state"` are not allowed.

`:variables` ({⟨**variable-slot**⟩}$^+$) **[optional]**

Definition of the variables of the module.

Variables are all named numerical entities that can be used in mathematical expressions. Like terminals the variables are defined as parts of a module. Excerpt form Section 1.3.2:

```
12        :variables (("n" :is-a "variable"
13        :system-theoretic "state"
14        :value "100.0d0"
15        :minimum "0.0d0"
16        :maximum "200.0d0"
17        :unit "kmol"
18        :report "all"
19        :documentation "molar holdup")
20        ("jen" :system-theoretic "input"
21        :value "10.0d0")
22        ...)
23
```

Within `:variables` all states, inputs, outputs, intermediates (help-variables) parameters and indices are defined that occur in the model of (8). In the example the variables `n` are defined as a state and `jen` as an input of the module `tank` respectively. In general the definition of a single variable looks like this:

⟨variable-slot⟩ :== (⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties` ({⟨property-slot⟩}$^+$)]
[`:indices` ({⟨index-definition⟩}$^+$)]
[`:system-theoretic` `"state"` | `"input"` | `"output"` | `"help"` |
`"structure-parameter"` | `"real-parameter"` |
`"integer-parameter"` | `"index"`]
[`:positive-vector-direction` `"in"` | `"out"`]
[`:value` ⟨math-expression⟩ | ⟨array-of-math-expressions⟩]
[`:minimum` ⟨math-expression⟩]
[`:maximum` ⟨math-expression⟩]
[`:relative-error` ⟨math-expression⟩]
[`:unit` ⟨math-unit-expression⟩]
[`:report` `"all"`]
)

The following part options may be used:

`:is-a` ⟨**class-name**⟩ [**optional**]
This defines the class of the variable. This option is optional and the rules mentioned above for the terminals apply here too.

The attribute options `:documentation`, `:properties`, `:indices`, `:system-theoretic`, `:positive-vector-direction`, `:report`, `:value`, `:minimum`, `:maximum`, `:relative-error` und `:unit` correspond to the command options of `define-variable`, that is described in more detail in Section 5.1. In the case of `n` a very explicit description of a variable is practiced, so nearly all applicable optional options are used. Instead the definition of `jen` only specifies the essential and non-optional things.

`:system-theoretic` `"state"` | `"input"` | `"output"` | **...** [**optional**]
Assignment of a system-theoretic type to the variable, `n` is defined as a state of the module.

`:value` ⟨**math-expression**⟩ | ⟨**array-of-math-expressions**⟩ [**optional**]
Definition of a value for the variable. This value is a mathematical expression. The syntax of the mathematical expressions is almost the same as in the CodeGenerator-language [Kö00], and is explained in Section **??**. For a state variable the `:value` denotes the initial value.

`:minimum` ⟨**math-expression**⟩ [**optional**]
Definition of a minimal value that the variable may reach during simulation.

`:maximum` ⟨**math-expression**⟩ [**optional**]
Definition of a maximal value that the variable may reach during simulation.

`:relative-error` ⟨**math-expression**⟩ [**optional**]
Definition of the numerical that is assumed for that variable.

`:unit` ⟨**math-unit-expression**⟩ [**optional**]
Definition of a physical unit for the variable. In the actual version of ProMoT this is only used for documentation purposes.

`:report` `"all"` [**optional**]
This denotes that the variable `n` should be written as a simulation result from Diva.

`:equations` ({⟨**equation-slot**⟩}$^+$) **[optional]**

Definition of the equations of a module.

```
50        :equations (("bal_n" :is-a "equation"
51        :relation ":diff<t> n == jen - jan"
52        :documentation "balance for the molar holdup")
53        ("fd_out" :is-a "equation"
54        :relation "0.0d0 == jan*jan - aa*aa * 2.0d0 * g * c * n/a"
55        :documentation "torricellis law for the outlet"))
56
```

The general form for the definition of a single equation looks like:

⟨equation-slot⟩ :== (⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties` ({⟨property-slot⟩}$^+$)]
[`:indices` ({⟨index-definition⟩}$^+$)]
[`:relation` ⟨math-relation⟩]
[`:variables` ({⟨weak-slot⟩}$^+$)]
)

The following attribute may be used:

`:is-a` ⟨**class-name**⟩ **[optional]**

Using a predefined equation-class for this equation. The given class must be a subclass of `equation`. If this option is omitted, the class of a possibly overridden, inherited equation in a superclass (with the same name) will be used, or the class `equation` itself.

`:documentation`, `:properties`, `:indices` **and** `:relation`

are described in Section 5.2 and are equivalent to the options of the command `define-equation`.

Mathematical Model: As described above an elementary module is aggregated from terminals, variables and equations. The system border of the module is described by its terminals and it exchanges through the terminals matter, momentum, energy and information with other modules. Every terminal contains a set of variables that occur in the $\mathbf{x}$, $\mathbf{u}$, $\mathbf{p}$, $\mathbf{h}$ or $\mathbf{y}$ of the complete systems model (see Equation 1). In contrast to the directed couplings without retroactivity in DIVA the coupling of modules within PROMOT may be direction-free and retroactive.

PROMOT generates a mathematical model out of the MDL description of the module that has a similar form as the apparatus model (1).

$$\begin{array}{lrcll}
\text{DA-System} & \mathbf{B}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t)\cdot\dot{\mathbf{x}} & = & \mathbf{f}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) & ;t > t_0 \\
\text{help variable assignments} & \mathbf{h} & := & \mathbf{g}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) & ;t > t_0 \\
\text{initial conditions} & \mathbf{x}(t_0) & = & \mathbf{x}_0 & \\
\text{output assignments (mapping out of } \mathbf{x}) & \mathbf{y} & := & \mathbf{C}\cdot\mathbf{x} & ;t \geq t_0
\end{array} \tag{8}$$

A coupled module is usually composed out of i_0 modules. The mathematical model of the module contains the following vectors of variables:

- state variables $\mathbf{x} \in \mathcal{R}^{n_x}$
- input variables $\mathbf{u} \in \mathcal{R}^{n_u}$;
- output variables $\mathbf{y} \in \mathcal{R}^{n_y}$;
- help variables (intermediate variables) $\mathbf{h} \in \mathcal{R}^{n_h}$;
- model parameters that are constant in time $\mathbf{p} \in \mathcal{R}^{n_p}$.

The following rules apply for the left hand side matrix $\mathbf{B}$, the right hand side function $\mathbf{f}$ and the vector of help variables:

- $\mathbf{B} \in \mathcal{R}^{n \times n_x}$;
- $\mathbf{f} \in \mathcal{R}^n$ (n is the number of equations and state variables);
- $\mathbf{g} \in \mathcal{R}^{n_h}$.

The intermediate variables are calculated by explicit assignments. The user has to provide a model that does not contain algebraic cycles in the help variable assignments (an example for such cycle is $a := 2b \quad b := c + d \quad d := 3a + b$). If unsure about the properties of the connected system, one should use algebraic equations and state variables instead. ProMoT optimizes the complete equation system and transforms algebraic equations to explicit assignments, if possible.

4.1.2 Composed Modules

Description in Mdl: Composed modules are defined like elementary modules with `define-module`. In this command two new options can be used: `:modules` and `:links`. As an example a cascade of two liquid tanks is described. This cascade is defined by the following Mdl command as the module `two-tanks`.

```
1   (define-module :class "two-tanks"
      :super-classes ("module")
      :documentation "Kaskade zweier Reinstoffbehaelter"
      :modules (("t1" :is-a "tank"
5                     :variables (("n" :value "200")))
                ("t2" :is-a "tank"))
      :links (("t1_t2" :terminals( "t1.out" "t2.in")))
      :terminals (("in" :is-eq-to  "t1.in")
                  ("out" :is-eq-to "t2.out")))
```

Two instances (`t1`, `t2`) of the elementary module `tank` are aggregated to the composed module `two-tanks`. The module tank is used in this command as a part of another module but remains itself unchanged. It is still possible to write out a device model of a single tank to Diva. The submodules get names (`t1`, `t2`) inside the context of the whole composed module. The molar holdup of the submodule `t1` is parametrized with an initial value 200.

The command options `:modules` and `:links` of the command `define-module` are defined as follows:

`:modules` ($\{\langle$**module-slot**$\rangle\}^+$) **[optional]**

Definition of the submodules of a composed module.

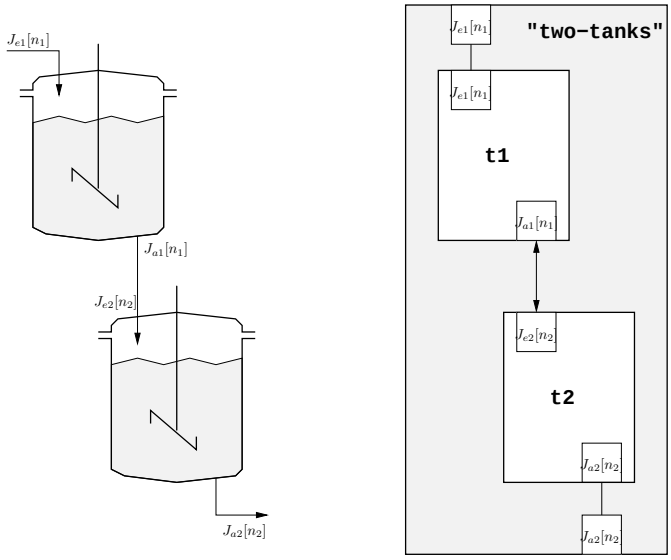


Figure 5: Flowchart of a cascade with two tanks. This cascade consists of two modules of the type `tank`. The outlet of the upper tank is coupled with the inlet of the lower one. The inlet of the cascade is the inlet of the upper tank and the outlet is the outlet of the lower tank.

⟨module-slot⟩ :== `(` ⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:terminals (` {⟨weak-slot⟩}$^+$ `)`]
[`:variables (` {⟨slot⟩}$^+$ `)`]
`)`

`:links` `(` {⟨**link-slot**⟩}$^+$ `)` **[optional]**

Definition of the coupling relations between the terminals of the different submodules of a composed module.

⟨link-slot⟩ :== `(` ⟨slot-name⟩ `:terminals (`⟨slot-path⟩`)`
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
`)`

The compatibility of coupled terminals and the generation of coupling equations is discussed in Section 6.2.

Mathematical Model: A composed module is described by a mathematical model that has a similar form as the model (8). In a composed module this model consists of the models of the

J submodules and the coupling equations connecting the different submodules:

$$
\begin{array}{lrcll}
\text{DA-System} & \underbrace{\begin{pmatrix} \mathbf{B}_0(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \mathbf{B}_1(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \dots \\ \mathbf{B}_J(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \mathbf{0} \end{pmatrix}}_{\mathbf{B}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t)} \cdot \begin{bmatrix} \dot{\mathbf{x}_0} \\ \dot{\mathbf{x}_1} \\ \dots \\ \dot{\mathbf{x}_J} \\ \dot{\mathbf{x}_K} \end{bmatrix} & = & \underbrace{\begin{pmatrix} \mathbf{f}_0(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \mathbf{f}_1(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \dots \\ \mathbf{f}_J(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \mathbf{K}_1 \cdot \mathbf{x} - \mathbf{K}_2 \cdot \mathbf{x} \end{pmatrix}}_{\mathbf{f}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t)} & ; t > t_0 \\
\text{Intermediate assignments} & \underbrace{\begin{pmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \\ \dots \\ \mathbf{h}_J \end{pmatrix}}_{\mathbf{h}} & := & \underbrace{\begin{pmatrix} \mathbf{g}_0(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \mathbf{g}_1(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \\ \dots \\ \mathbf{g}_J(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \end{pmatrix}}_{\mathbf{g}(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t)} & ; t > t_0 \\
\text{Initial Conditions} & \mathbf{x}(t_0) & = & \mathbf{x}_0 & \\
\text{Output Assignments} & \mathbf{y} & := & \mathbf{C} \cdot \mathbf{x} & ; t \geq t_0
\end{array}
\tag{9}
$$

The DAE of the modularized model (9) consists of $J + 2$ partial systems:

- the direct system

  $$\mathbf{B}_0(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \cdot \mathbf{x} = \mathbf{f}_0(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t)$$

  contains the equations that are directly contained in the top-level of the composed module. This equations may determine global quantities that apply to all of the submodules.

- the subsystems

  $$\mathbf{B}_j(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \cdot \mathbf{x} = \mathbf{f}_j(\mathbf{x},\mathbf{h},\mathbf{u},\mathbf{p},t) \qquad ; j = 1,2,\dots,J$$

  are contained in the J submodules.

- the coupling system

  $$\mathbf{0} = \mathbf{K}_1 \cdot \mathbf{x} - \mathbf{K}_2 \cdot \mathbf{x}$$

  contains the coupling equations that connect the interface variables in the connected terminals of the J submodules.

The variable vectors $\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{h}$ and $\mathbf{p}$ are the union of all variable vectors of the submodules and the direct variables of the composed module.

Although in general the variable sets of the submodules are disjunct, it is possible to introduce "shared" variables. This makes sense if, for instance the temperature of a composed module is calculated by a global balance, but applies to each of the submodules. This can be defined in the way that that the temperature variable is a strong part of composed module and every submodule uses this variable as a weak part (by `:is-eq-to`). Another example may be, that the same set of parameters is used in different submodules, but is subject of parameter estimation. In this case one needs only one instance of the parameter set in the simulation environment and should also define the parameters as strong parts globally and propagate them to the submodules as weak parts. Only strong parts appear directly in the variable vectors. Weak parts, that are used in equations or links are internally substituted by the strong parts, they refer to.

In the example model (Figure 5) the resulting variable set is the following is the union of the variables of the submodules `t1` and `t2`:

$$\begin{array}{lcl} \mathbf{x} & = & \begin{pmatrix} n_1 & J_{a1}[n] & J_{e2}[n] & n_2 & J_{a2}[n] \end{pmatrix}^T \\ \mathbf{u} & = & \begin{pmatrix} J_{e1}[n] \end{pmatrix}^T \\ \mathbf{y} & = & \begin{pmatrix} J_{a2}[n] \end{pmatrix}^T \\ \mathbf{h} & = & \begin{pmatrix} \end{pmatrix} \\ \mathbf{p} & = & \begin{pmatrix} g_1 & c_1 & A_1 & A_{a1} & g_2 & c_2 & A_2 & A_{a2} \end{pmatrix}^T \end{array} \tag{10}$$

A variable that is defined as input in the submodule k becomes a state in the composed module if it is not contained in outer terminals of the composed module (9). This means that this variable is only contained in terminals that are internally connected to other terminals, and a coupling equation exists, that determines this input variable. This is the case for the input flux $J_{e2}[n]$ in the example of Figure 5. The input $J_{e1}[n]$ of `t1` and the output $J_{a2}[n]$ von `t2` become an input and an output of the composed module respectively. A special case, that arises for indexed input variables, is described below.

4.1.3 Indexed Modules

Description in Mdl: Indexed modules are defined like the aggregated modules with `define-module` and the option `:modules`. Additionally the option `:indices` is used. As an example the module `tank` shall be used in an indexed form to describe a cascade of tanks `array-of-tanks` with arbitrary, finite size:

```
1   (define-module :class "array-of-tanks"
      :super-classes ("module")
      :documentation "Cascade of tanks"
      :variables (("nt" :is-a "variable"
5                       :system-theoretic "structure-parameter"
                        :value "3"
                        :documentation "Number of tanks")
                  ("i" :is-a "variable"
                       :system-theoretic "index"))
10    :modules (("ta" :is-a "tank"
                      :indices ((:index "i" :lower "1" :upper "nt"))))
      :links (("tl" :indices ((:index "i" :lower "1" :upper "nt-1"))
                    :terminals( "ta[i].out" "ta[i+1].in")))
      :terminals (("in" :is-eq-to  "ta[1].in")
15                ("out" :is-eq-to "ta[nt].out")))
```

In this example the index `i` is defined which runs from the lower bound `"1"` to the upper bound `"nt"`. The submodule `ta` represents the one-dimensional array of `"nt"` single tanks. In the `:links` the `nt` indexed submodules are connected by `nt-1` links. The meaning is the following: Every terminal `"out"` of a tank `"ta[i]"` (`"ta[i].out"`) is connected with the terminal `"in"` of the next tank `"ta[i+1]"`. The outer terminals of the `array-of-tanks` are defined as the input of the first tank (`ta[1].in`) and the output of the last one (`ta[nt].out`).

4.1.4 Diva specific definitions

ProMoT acts currently as a modeling tool for Diva. When a module is transferred from ProMoT to Diva, a number of files are written. The names for these files can be specified,

using `mdl2diva` or the model output dialog (see Section 7.7). For the sake of convenience these options can be stored with the module in MDL. The model output dialog sets these attributes automatically and uses them together with `mdl2diva` as default values. The different options get explained here:

`:diva-generation-directory` Specifies the directory, where the model with all the following files should be written to. The default is the current directory of PROMOT.

`:diva-cg-file-name` Defines the name of the CODEGENERATOR input file that is written by PROMOT. It is located in the generation directory.

`:diva-fortran-files-subname` specifies the identifying part of the names of the FORTRAN files for DIVA. The default is `ab01`. This part of the name may be 4 characters long and is used for the FORTRAN subroutine files: If the name part was `ab01` the FORTRAN files are called `dab01i.f`,`dab01f.f`,`dab01a.f` and `dab01p.f`. This files are written by the CODEGENERATOR.

`:diva-data-file-name` Into the file with this name the CODEGENERATOR writes the parameter and initial values for the simulation model. The name must end in ".dat" and the prefix can be at most 5 characters long and may not start with c. This file appears again in the `.anl` file.

`:diva-latex-file-name` Into this file the CODEGENERATOR writes LaTeX code to document the complete model. This documentation contains tables of all variables and equations of the model.

`:diva-anl-file-name` This file is generated by PROMOT for the sake of simplicity. It contains a DIVA plant description for the case that a closed PROMOT model is written to DIVA. In this file the FORTRAN subroutines named by the `:diva-fortran-files-subname` and the block data in the `:diva-data-file-name` get connected. This file can be immediately used for closed models in the command `inianl` in DIVA.

4.2 Terminals

Terminals describe the system interface of modules. A terminal defines a logical collection (set) of variables that should be connected as a unit in one step. This set of variables is a subset of the variables of the modules itself. When coupling a terminal to a terminal of another module, coupling equations are established. Through the terminals, the inner state of a module is made accessible to the outside.

The modeling-entity `terminal` is a direct subclass of the `structural-modeling-entity`. A terminal(-type) is defined by the MDL-command `define-terminal` . The elements in the list of `:super-classes` (Section 3) must be subclasses of `terminal` or the class `terminal` itself.

The general form of `define-terminal` is defined by the following EBNF Rule:

⟨define-terminal⟩ :== `(define-terminal :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:geometry-side "top"` |`"bottom"` |`"left"`|`"right"`]
[`:geometry-position` {⟨property-slot⟩}$^+$]
[`:link-color` {⟨color-name⟩}$^+$]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:variables (` {⟨variable-slot⟩}$^+$ `)`]
`)`

In addition to the general options described in Section 3, the option `:variables` may be used to describe variables that are contained in this terminal. Internally in the ProMoT system a terminal is a container of variables.

`:variables` `(` {⟨**variable-slot**⟩}$^+$ `)` **[optional]**
Definition of the variables of a terminal.

There is a distinction between

- elementary terminals and
- indexed terminals.

4.2.1 Elementary Terminals

For the definition of the terminals in the example models `tank` (Section 1.3.2) and `two-tanks` (Section 4.1.2) the command `define-terminal` is not used. But the option `:terminals` is used to define the elementary terminals of these modules. In the option `:variables` the variables for the molar input and output flux are assigned as weak parts of the respective terminals. The important point in this assignment is the naming of the variables. While assigning the variable `"jen"` and `"jan"` to the terminals, the variables get a name that is valid for them in the context of the terminal. These names are necessary for the generation of coupling equations, since the variables with the same name across all linked terminals are coupled with these equations. Considering a pair of terminals `"a"` and `"b"` which both have the variables `"j"` and `"k"` the coupling equations $a.k == b.k$ and $a.j == b.j$ are established based on the names of the variables.

4.2.2 Indexed Terminals

While using indexed terminals it becomes possible to parameterize the number of terminals of a module. In the following a liquid tank with n_e inlets and n_a outlets should be investigated (Figure 6). All inlets are located at the top of the tank and all outlets are at the bottom but

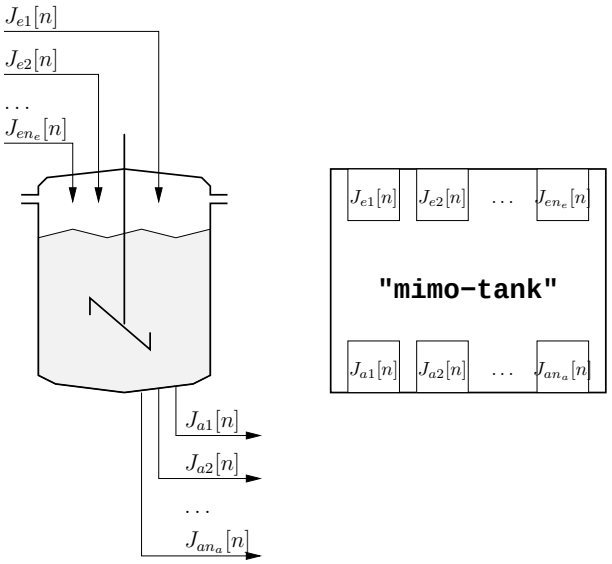


Figure 6: The flowchart of a tank with n_e inlets and n_a outlets called `mimo-tank` .

may have individual cross sections A_{ai}. The mathematical model (7) is changed as follows:

$$\begin{aligned} \frac{dn}{dt} &= \sum_{i=1}^{n_e} J_{ei}[n] - \sum_{i=1}^{n_a} J_{ai}[n] && ;t > t_0 \\ 0 &= J_{ai}[n]^2 - A_{ai}^2 \cdot 2 \cdot \frac{gcn}{A} && ;t \geq t_0, \quad i = 1, \ldots, n_a \\ n(t_0) &= n_0 \end{aligned} \tag{11}$$

Both of the indexed terminals `vector-of-inlets`, `vector-of-outlets` and the module `mimo-tank` can be described in MDL like:

```
(define-terminal :class "vector-of-inlets"
  :super-classes ("terminal")
  :documentation "Vektor von Zulaeufen"
  :indices ((:index "i" :lower "1" :upper "nt"))
  :variables (("nt" :is-a "variable"
                    :system-theoretic "structure-parameter")
              ("i" :is-a "variable"
                   :system-theoretic "index")
              ("jn" :is-a "variable"
                    :system-theoretic "input"
                    :value "10.0d0")))

(define-terminal :class "vector-of-outlets"
  :super-classes ("terminal")
```

```
15     :documentation "Vektor von Ablaeufen"
       :indices ((:index "i" :lower "1" :upper "nt"))
       :variables (("nt" :is-a "variable"
                         :system-theoretic "structure-parameter")
                   ("i" :is-a "variable"
20                      :system-theoretic "index")
                   ("jn" :is-a "variable"
                         :system-theoretic "output"
                         :report "all"
                         :value "10.0d0")))
25
     (define-module :class "mimo-tank"
       :super-classes ("module")
       :documentation "Reinstoffbehaelter mit beliebig vielen Zu- und Ablaeufen"
       :terminals (("in" :is-a "vector-of-inlets"
30                       :variables (("nt" :is-eq-to "parent.ne")))
                   ("out" :is-a "vector-of-outlets"
                          :variables (("nt" :is-eq-to "parent.na"))))
       :variables (("ne" :is-a "variable"
                         :system-theoretic "structure-parameter"
35                       :value "2"
                         :documentation "Anzahl Zulaeufe")
                   ("na" :is-a "variable"
                         :system-theoretic "structure-parameter"
                         :value "3"
40                       :documentation "Anzahl Ablaeufe")
                   ("i" :is-a "variable"
                        :system-theoretic "index")
                   ("n" :is-a "variable"
                        :system-theoretic "state"
45                      :value "100.0d0"
                        :minimum "0.0d0"
                        :maximum "200.0d0"
                        :unit "kmol"
                        :report "all"
50                      :documentation "Molinhalt")
                   ("a" :is-a "variable"
                        :system-theoretic "real-parameter"
                        :value "1.0d0"
                        :unit "m^3"
55                      :documentation "Grundflaeche")
                   ("aa" :is-a "variable"
                         :indices ((:index "i" :lower "1" :upper "na"))
                         :system-theoretic "real-parameter"
                         :value ("0.05d0" "0.1d0" "0.2d0")
60                       :unit "m^3"
                         :documentation "Querschnittsflaechen der Ablaeufe")
                   ("c" :is-a "variable"
                        :system-theoretic "real-parameter"
                        :value "55.0d0"
65                      :unit "kmol/m^3"
                        :documentation "molare Konzentration von Wasser")
                   ("g" :is-a "variable"
```

```
                    :system-theoretic "real-parameter"
                    :value "9.81d0"
                    :unit "m/s^2"
                    :documentation "Erdbeschleunigung"))
    :equations (("bal_n" :is-a "equation"
                         :relation ":diff<t> n == :sum<i,1,ne> in[i].jn
                                                - :sum<i,1,na> out[i].jn"
                         :documentation "Molmengenbilanz")
                ("fd_out" :is-a "equation"
                          :indices ((:index "i" :lower "1" :upper "na"))
                          :relation "0.0d0 == out[i].jn*out[i].jn -
                                              aa[i]*aa[i] * 2.0d0 * g * c * n/a"
                          :documentation "fluiddynamische Beziehung fuer die
                                          Ablaufstroeme: Toricelli")))
```

In this case the terminals are also the strong containers of the variables, i.e. the variables are not directly contained by the module. Inside the equations the variables are referenced via a hierarchical indexed slot-path : `in[i].jn` and `out[i].jn`.

5 Behavioral modeling entities

The class `behavioral-modeling-entity` is the superclass of all variables, equations, transitions and places. For discrete modeling the `transition` and `place` are subsumed under `discrete-modeling-entity` whereas the `variable` and `equation` are subclasses of `continuous-modeling-entity`. This section is divided into three parts: in the first one is explained, how scalar variables can be defined with the command `define-variable`. The second part deals with the definition of equations with `define-equation`. Finally, in the third part it will be shown in an example how indexed variables and equations can be used. For the definition of `tank` (Section 1.3.2) and `two-tanks` (Section 4.1.2) the commands `define-variable` and `define-equation` have not been used. Instead the variables and equations inside `tank` have been defined implicitly as instances of `variable` and `equation` with some local specializations.

Alternatively and in the same manner a modeler can predefine classes of variables and equations with common attributes. Therefore the commands `define-equation` and `define-variable` are necessary. With this method it is possible, to define meaningful variable types for often used physical quantities and common equations and to store them in a modeling library. A definition that is equivalent to `tank` looks like the following:

```
(define-variable :class "molar-holdup"
  :super-classes ("variable")
  :value "100.0d0"
  :minimum "0.0d0"
  :maximum "200.0d0"
  :unit "kmol"
  :documentation "molar holdup")

(define-variable :class "molar-concentration"
  :super-classes ("variable")
  :unit "kmol/m^3"
  :documentation "molar concentration")

(define-variable :class "molar-flux"
  :super-classes ("variable")
  :value "10.0d0"
  :unit "kmol/h"
  :documentation "molar flow")

(define-variable :class "area"
  :super-classes ("variable")
  :unit "m^3"
  :documentation "area")

(define-variable :class "acceleration-of-gravity"
  :super-classes ("variable")
  :system-theoretic "real-parameter"
  :value "9.81d0"
  :unit "m/s^2"
  :documentation "gravity constant")

(define-equation :class "material-balance"
  :super-classes ("equation")
  :relation ":diff<t> n == jen - jan"
```

```
35     :documentation "material balance")

     (define-equation :class "toricelli"
       :super-classes ("equation")
       :relation "0.0d0 == jan*jan - aa*aa * 2.0d0 * g * c * n/a"
40     :documentation "torricellis law for the outlet")

     (define-module :class "tank-3"
       :super-classes ("module")
       :documentation "Pure substance tank"
45     :terminals (("in" :is-a "terminal"
                    :variables (("jn" :is-eq-to "parent.jen")))
                   ("out" :is-a "terminal"
                    :variables (("jn" :is-eq-to "parent.jan"))))
       :variables (("n" :is-a "molar-holdup"
50                   :system-theoretic "state"
                     :report "all")
                   ("jen" :is-a "molar-flux"
                    :system-theoretic "input")
                   ("jan" :is-a "molar-flux"
55                   :system-theoretic "output")
                   ("a" :is-a "area"
                    :system-theoretic "real-parameter"
                    :value "1.0d0"
                    :documentation "cross section of tank")
60                 ("aa" :is-a "area"
                    :system-theoretic "real-parameter"
                    :value "0.1d0"
                    :documentation "cross section of outlet")
                   ("c" :is-a "molar-concentration"
65                  :system-theoretic "real-parameter"
                    :value "55.0d0"
                    :documentation "molar concentration of water")
                   ("g" :is-a "acceleration-of-gravity"))
       :equations (("bal_n" :is-a "material-balance"
70                   :variables (("n" :is-eq-to "parent.n")
                                 ("jen" :is-eq-to "parent.jen")
                                 ("jan" :is-eq-to "parent.jan")))
                   ("fd_out" :is-a "toricelli"
                    :variables (("a" :is-eq-to "parent.a")
75                              ("aa" :is-eq-to "parent.aa")
                                ("c" :is-eq-to "parent.c")
                                ("g" :is-eq-to "parent.g")
                                ("n" :is-eq-to "parent.n")
                                ("jan" :is-eq-to "parent.jan")))))
```

In this example the variables `molar-holdup`, `molar-concentration`, `molar-flux`, `area` and `acceleration-of-gravity` are defined as direct subclasses of `variable` and used as parts of the module `tank-3`. The equations `material-balance` and `torricelli` are defined as subclasses of `equation` and also used via `::is-a` inside of `tank-3`. This variables and equations can also be used as parts of other modules.

5.1 Variables

Variables describe together with the equations the continuous mathematical model of a module. For the definition of variable(-classes) the MDL command `define-variable` can be used. With these class definitions a semantic hierarchy of variable-classes like `concentration`, `molar-holdup`, `heat-flux` etc. can be specified. Besides this variables are classified by their system-theoretic type and their vectorial direction. It is possible to use variables as scalars or in an indexed form (arrays).

5.1.1 System-theoretic type

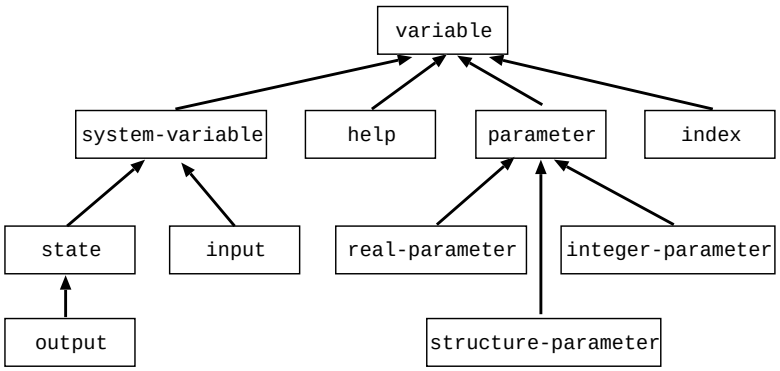


Figure 7: Specialization hierarchy of system-theoretic types.

In a fully specified model, every variable must have a system-theoretic type. The available system-theoretic types of PROMOT are the same as in the CODEGENERATOR and in DIVA. The system-theoretic type determines the meaning of the variable for the simulation and optimization algorithms. It used to determine, to which variable vector of $\mathbf{x}, \mathbf{u}, \mathbf{y}, \mathbf{h}$ and $\mathbf{p}$ the variable belongs to.

What is the meaning of these types? State-, input- and output-variables or system-variables represent the state of the module that may change during simulation. Input-variables are determined outside of a module and must be connected to another model, output variables are state variables, that are definitely calculated by the equations of a module and may be used in the terminals. State variables represent the inner state of a module. They can also be used in the terminals, but in this case it is not specified, whether they are determined by the equations of this module or they are a degree of freedom. The final determination is established through the connection of the terminals.

Parameters are values that do not change during simulation. Real-parameters act usually as constant physical quantities in the model. They may be calculated in parameter estimation and optimization tasks. Integer-parameters can be used like real-parameters with an integer value. They are also often used as switches in conditional expressions. Real- and integer-parameters may be changed on the DIVA command prompt with commands like `changeblo`. Structure-parameters are a special kind of parameters that act as special numbers for indices. They are used to determine the run length of different indices in equations and arrays consistently. They may not be changed in DIVA.

attribute `:system-theoretic`	system-theoretic type	variable vektor
`"state"`	`state`	**x**
`"input"`	`input`, `state`	**u**, **x**
`"output"`	`output`, `state`	**y**, **x**
`"structure-parameter"`	`structure-parameter`	**p**
`"integer-parameter"`	`integer-parameter`	**p**
`"real-parameter"`	`real-parameter`	**p**
`"help"`	`help`	**h**
`"index"`	`index`	—

For input and output variables the system-theoretic attribute in the model definitinon is not necessarily used during the simulation, but it depends on the links the module is connected with. If input or output variables only appear in terminals that are connected internally in the ProMoT model, these variables become state variables during simulation. Only if these variables are present in terminals of the top-level module, they become inputs and outputs and can be used to connect this module with other Diva models during simulation (see Section 4.1.2).

The system-theoretic types in ProMoT are realized by a class hierarchy like the modeling entities. That is assigned as a delegate to the variables. However, modelers can not change or directly access these classes. Different from the syntax in CodeGenerator input files, index variables in Mdl have to be declared explicitly in the list of `:variables`.

5.1.2 Vectorial type

Variables that are contained in terminals, can be defined either as vectorially undirected, potential quantities or vectorially directed, flux quantities.

vectorially directed variables: A vectorially directed variable represents a physical quantity that is a vector for which a positive direction has to be defined in the model. In general these variables are fluxes or gradients of potentials in the model. The definition of the direction takes place with respect to a balanced element (e.g. volume). In ProMoT this balanced element is the module, the variable belongs to. Depending on the definition, the variable is used with a positive or negative sign in the equations. Examples for this kind of variables are molar flows, forces, torques and electric currents. The modeler can define the variables as `:positive-vector-direction "in"` or `:positive-vector-direction "out"`. This definitions allow to specify if a positive value of the flux is directed into the module (`"in"`) or out of the module (`"out"`). This changes nothing on the balance equations of the module. But if this variables get connected to other variables, the sign for the variables in the coupling equations is determined by this attribute (see Section 6.2).

vectorially undirected variables: are potentials in the physical sense that do not have a vectorial direction. Examples for physical quantities of this kind are temperature, pressure, molar fraction and electrical potential

For the description of the `tank` in Section 1.3.2 only vectorially undirected variables were used. An alternative description is the following, where the molar fluxes $J_e[n]$ and $J_a[n]$ are specified as positive inwards directed variables. Therefore the model equations of the tank have to be changed as follows:

$$\begin{aligned} \frac{dn}{dt} &= J_e[n] + J_a[n] && ; t > t_0 \\ 0 &= (-J_a[n])^2 - A_a^2 \cdot 2 \cdot \frac{gc}{A} && ; t \geq t_0 \end{aligned} \tag{12}$$

The output flow $J_a[n]$ now appears with a positive sign in the mass balance and the calculation of the outlet flow. The result of this are the following changes to the lines 20-29 of the MDL description of `tank`.

```
19    ...
20                  ("jen" :is-a "variable"
21                         :system-theoretic "input"
22                         :positive-vector-direction "in"
23                         :value "10.0d0"
24                         :unit "kmol/h"
25                         :documentation "molar inflow")
26                  ("jan" :is-a "variable"
27                         :system-theoretic "output"
28                         :positive-vector-direction "in"
29                         :value "-10.0d0"
30                         :unit "kmol/h"
31                         :documentation "molar outflow")
32    ...
33
```

The new equations are in lines 50-57:

```
49    ...
50      :equations (("bal_n" :is-a "equation"
51                           :relation ":diff<t> n == jen + jan"
52                           :documentation "mass balance")
53                  ("fd_out" :is-a "equation"
54                            :relation "0.0d0 == (-jan)*(-jan) -
55                                                aa*aa * 2.0d0 * g * c * n/a"
56                            :documentation "Torricelli Equation")))
57
```

When two tanks of this type are coupled to a cascade the following coupling equation is established between $J_{a1}[n]$ and $J_{e2}[n]$:

$$0 = J_{a1}[n] + J_{e2}[n] \tag{13}$$

The values of the outflows $J_{a1}[n]$ and $J_{a2}[n]$ are always negative.

5.1.3 Variable descriptions in Mdl

A subclass of the modeling entity `variable` is defined by the command `define-variable`. This command has the attributes `:class`, `:super-classes`, `:documentation`, `:properties`, `:model-name` and `:indices` as explained in Section 3. Additionally the options `:value`, `:minimum`, `:maximum`, `:absolute-error` (`:relative-error`), `:unit`, `:system-theoretic`, `:positive-vector-direction` and `:report` can be specified. Some of these options have been explained in Section 4.1.1. The general form of `define-variable` is described by the following EBNF Rule:

⟨define-variable⟩ :== `(define-variable :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]

```
[:system-theoretic "state" | "input" | "output" | "help" | "structure-parameter" |
                   "real-parameter" | "integer-parameter" | "index" ]
[:positive-vector-direction "in" | "out"]
[:value ⟨math-expression⟩ | ⟨array-of-math-expressions⟩]
[:minimum ⟨math-expression⟩]
[:maximum ⟨math-expression⟩]
[:relative-error[8] ⟨math-expression⟩]
[:absolute-error ⟨math-expression⟩]
[:unit ⟨math-unit-expression⟩]
[:report "lumped"|"distributed"|"all"|"none"]
[:variables ( {⟨variable-slot⟩}+ )]
)
```

`:system-theoretic` `"state"` | `"input"` | `"output"` | `"help"` | `"structure-parameter"` | `"real-parameter"` | `"integer-parameter"` | `"index"` **[optional]**

Definition of the system-theoretic type according to the table in Section 5.1.1.

For every variable this type has to be defined either locally or in a superclass. It is possible (from ProMoT version 0.6.3) to override the type locally.

`:positive-vector-direction` `"in"` | `"out"` **[optional]**

Definition of the vectorial type of an interface variable according to the Section 5.1.2.

Without this attribute the variable is a potential variable (i.e. has no vectorial direction). With `"in"` it becomes a positive inwards vectorial variable and with `"out"` it is a positive outwards vectorial variable.

`:value` ⟨**math-expression**⟩ | ⟨**array-of-math-expressions**⟩ **[optional]**

Definition of the (possibly initial) value of the variable.

The value is given as a mathematical expression or as an array of mathematical expressions. The syntax of the mathematical expressions in Mdl is similar to the syntax of the CodeGenerator input language [Kö00]. For input, output or state-variables the `:value` specifies the initial value of the variable. The definition of a value as ⟨array-of-math-expressions⟩ is not allowed for structure-parameters and help variables. The value of an indexed help variable should be a math expression that depends on the value of the index variables. It may contain `:cond()` statements. Indices should not have a specified value. The dimension of a ⟨array-of-math-expressions⟩ must match the dimensions that are specified in the option `:indices` of the variable.

Values may be defined depending on the values of other variables (the value may contain a calculation based on other variables). The following dependencies are allowed for the different system-theoretic types:

integer parameter may depend on structure parameters and other integer parameters,

real parameters and system variables (input, output and state) may contain structure parameters, integer parameters and other real parameters,

help variables can depend on every system-theoretic type.

Since help variables may depend on each other, they are ordered automatically according to their dependencies. Since they are calculated during simulation via a direct assignment,

there must not be a circular dependency of help variables. If ProMoT reports an error for a help variable, you should transform this help variable into a state and the value into an algebraic equation.

In the example `tank` only simple numbers are used as values for the variables. The following example defines the parameter `aa` as one tenth of the parameter `a`. Further a help variable `rhs_bal_n` is introduced for the right hand side of the mass balance `bal_n`.

```
(define-module :class "tank-4"
  :super-classes ("module")
  :documentation "Pure substance tank"
  :properties (("aggregate_state" :one-of ("liquid" "gas")
                                  :value "liquid")
               ("spatial_resolution" :one-of ("distributed" "concentrated")
                                     :value "concentrated"))
  :terminals (("in" :is-a "terminal"
                    :variables (("jn" :is-eq-to "parent.jen")))
              ("out" :is-a "terminal"
                     :variables (("jn" :is-eq-to "parent.jan"))))
  :variables (("n" :is-a "variable"
                   :system-theoretic "state"
                   :value "100.0d0"
                   :minimum "0.0d0"
                   :maximum "200.0d0"
                   :unit "kmol"
                   :report "all"
                   :documentation "molar holdup")
              ("jen" :is-a "variable"
                     :system-theoretic "input"
                     :value "10.0d0"
                     :unit "kmol/h"
                     :documentation "molar inflow")
              ("jan" :is-a "variable"
                     :system-theoretic "output"
                     :value "10.0d0"
                     :unit "kmol/h"
                     :documentation "molar outflow")
              ("a" :is-a "variable"
                   :system-theoretic "real-parameter"
                   :value "1.0d0"
                   :unit "m^3"
                   :documentation "cross section of tank")
              ("aa" :is-a "variable"
                    :system-theoretic "real-parameter"
                    :value "0.1d0*a"
                    :unit "m^3"
                    :documentation "cross section of outlet")
              ("c" :is-a "variable"
                   :system-theoretic "real-parameter"
                   :value "55.0d0"
                   :unit "kmol/m^3"
                   :documentation "molar concentration of water")
              ("g" :is-a "variable"
                   :system-theoretic "real-parameter"
```

```
                        :value "9.81d0"
                        :unit "m/s^2"
                        :documentation "gravity constant")
                ("rhs_bal_n" :is-a "variable"
                             :system-theoretic "help"
                             :value "jen - jan"
                             :documentation "Right hand side of material balance"))
        :equations (("bal_n" :is-a "equation"
                             :relation ":diff<t> n == rhs_bal_n"
                             :documentation "total material balance")
                    ("fd_out" :is-a "equation"
                              :relation "0.0d0 == jan*jan -
                                                  aa*aa * 2.0d0 * g * c * n/a"
                              :documentation "torricellis law for the outlet")))
```

`:minimum` ⟨**math-expression**⟩ [**optional**]

Definition of a minimal value, that this variable may encounter during simulation.

The default value is `"-100.0"`.

`:maximum` ⟨**math-expression**⟩ [**optional**]

Definition of a maximal value.

The minimum and maximum can be used together with the absolute error for the error estimation and step width control of the numerical routines.

`:absolute-error` ⟨**math-expression**⟩ [**optional**]

Definition of the maximum absolute error tolerance for this variable.

The default value is `"1.0d-6"`. DIVA calculates the allowed error limit as $e_{rel} * \max|x|, max + e_{abs}$. e_{rel} is the globally defined relative error of the numerical algorithm, max is the maximum value defined for this variable, x is the actual value of the state variable and e_{abs} is the absolute error given for this state variable. A good modeling practice is to define the attributes `:minimum`, `:maximum` and `:absolute-error` for classes of variables that represent similar physical quantities and therefore have similar values and error limits.

`:unit` ⟨**math-unit-expression**⟩ [**optional**]

Definition of a physical unit of a variable.

The value is a mathematical expression consisting of basic unit names (e.g. `"m/s"` or `"1/s"`). Up to the actual PROMOT version this information is mainly used for documentation purposes and is written as part of the `:meaning` into the CODEGENERATOR input file and to the DIVA model code. It is planned to check the units in the model, but currently there is no such test implemented.

`:report` `"all"` |`"lumped"` |`"distributed"` |`"none"` [**optional**]

Output of this variable as a DIVA process value during simulation.

`:report` specifies, if and how the DIVA simulator should write the value of this variable to the output file. `"lumped"` and `"all"` are synonymous for compatibility reasons. If this is specified, every variable will be written as a single value to the pvkey of its module. If the `:report` is given as `"distributed"` and the variable is indexed, the variable can be plotted across its index value with the matlab plotting tool. The `:report` option can also be inherited from superclasses and writing variables to the output file can be locally switched off by specifying `:report "none"`. `:report` also inhibits, that a (help-)variable is removed from the model by the model optimizer.

`:dont-optimize "yes" |"no"` **[optional]**

Inhibit the optimization of this variable.

Before writing a model for the simulation environment, ProMoT optimizes the resulting equation system. During this process, some system-variables are changed into help-variables and some latent help variables are removed from the simulation model. The function of the mathematical model is not changed in this process, but the resulting model can be calculated more efficiently. However, if a modeler does not want these optimizations for certain variables, they can be marked as `:dont-optimize "yes"`.

This situation applies for instance, if a state variable is used in parameter fitting, but ProMoT transforms it into a help variable. Since Diva accepts only state variables as inputs for the standard objective function in parameter estimation, this variable has to remain as a state.

5.2 Equations

The modeling entity `equation` is the superclass of all equations. For the definition of a separate equation the command `define-equation` is used. Equations can be scalar or indexed as the variables.

5.2.1 Mdl description

A subclass of the modeling entity `equation` is defined by the MDL command `define-equation`. The options `:class`, `:super-classes`, `:documentation`, `:properties` and `:indices` can be specified in this command as explained in Section 3. Further the modeler can specify the options `:relation` and `:variables`. These options have been partially described before in Section 4.1.1.

The general form of `define-equation` is given by the following EBNF Rule:

⟨define-equation⟩ :== `(define-equation :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:relation` ⟨math-relation⟩]
[`:variables (` {⟨variable-slot⟩}$^+$ `)`]
`)`

`:relation` ⟨**math-relation**⟩ [**optional**]

Definition of the mathematical expression of the equation.

In contrast to a `:value` (see page 38), a `:relation` must contain a mathematical relation, i.e. a mathematical expression that contains one equal-sign (`"=="`). For differential equations the expression may contain sums of differential terms in the form ⟨unquoted-math-expression⟩ `*:diff<t>` ⟨variable-ref⟩
where ⟨variable-ref⟩ must refer to a state-variable. (see Appendix A). PROMOT transforms expressions of this form into the linear implicit form that can be handled by DIVA (see (1) in Section 1).

`:variables` `(` {⟨**variable-slot**⟩}$^+$ `)` [**optional**]

Assignment of variables as parts of the equation. This makes sense, if the equation has been specified outside the context of the module, and the names of the used variables do not match to those used in the module. In this case a mapping can be defined as a list of `:variables` using the option `:is-eq-to` (see example in the code of `tank-3` on page 33).

5.3 Indexed variables and equations

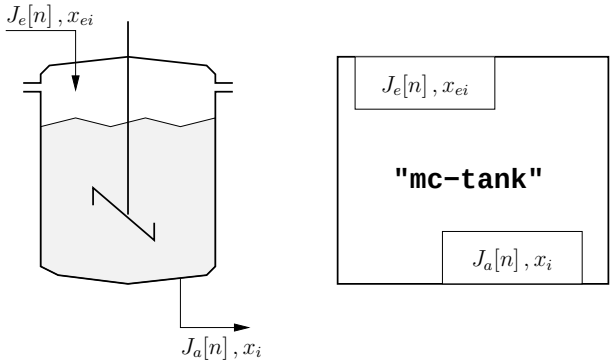


Figure 8: Flowchart of a single-phased multi-component tank with one inflow and one outflow.

In this section a multi-component tank should be considered as an example for the use of indexed variables and equations. The multi-component tank `mc-tank` is derived from the `tank` introduced in Section 1.3. As the `tank` it has one inflow and one outflow. But the `mc-tank` contains a mixture of N_c components. In its inflow the molar fractions x_{ei} of the input flow must be specified in addition to the already known molar flow $J_e[n]$. In the outlet also all molar fractions x_i of the mixture are present. In addition to the total mass balance and the Torricelli equation of the model (7), component mass balances are necessary for the partial molar holdups n_i, a definition of the molar fractions x_i and since there is an algebraic constraint on the molar fractions, a summation condition is used for the partial molar holdups. This results in the following mathematical model for the `mc-tank`:

$$\begin{aligned}
\frac{dn}{dt} &= J_e[n] - J_a[n] && ;t > t_0 \\
0 &= J_a[n]^2 - A_a^2 \cdot 2 \cdot \frac{gc}{A} && ;t \geq t_0 \\
n(t_0) &= n_0 \\
\frac{dn_j}{dt} &= J_e[n]\, x_{ej} - J_a[n]\, x_j && ;j = 1, \ldots, N_c - 1 \quad , t > t_0 \\
n_j &= n x_j && ;j = 1, \ldots, N_c \quad , t \geq t_0 \\
n &= \sum_{j=1}^{N_c} n_j && ;t \geq t_0 \\
n_j(t_0) &= n_{0j} && ;j = 1, \ldots, N_c
\end{aligned} \tag{14}$$

The `mc-tank` is defined as direct subclass of `tank`. Therefore `mc-tank` inherits all terminals, variables and equations of `tank`. Only the new parts must be described, and the terminals `in` and `out` have to be specialized:

```
1   (define-module :class "mc-tank"
      :super-classes ("tank")
      :documentation "Multi component tank"
      :terminals (("in"  :variables (("nc" :is-eq-to "parent.nc")
5                                    ("x"  :is-eq-to "parent.xe")))
                  ("out" :variables (("nc" :is-eq-to "parent.nc")
```

```
                                    ("x"  :is-eq-to "parent.x"))))
      :variables (("nc" :is-a "variable"
                        :system-theoretic "structure-parameter"
                        :value "3"
                        :documentation "number of components")
                  ("j" :is-a "variable"
                       :system-theoretic "index"
                       :documentation "index for components")
                  ("ni" :is-a "variable"
                        :system-theoretic "state"
                        :indices ((:index "j" :lower "1" :upper "nc"))
                        :value ("20.0" "30.0" "50.0")
                        :documentation "partial molar holdups")
                  ("x"  :is-a "variable"
                        :system-theoretic "output"
                        :indices ((:index "j" :lower "1" :upper "nc"))
                        :value ("0.2" "0.3" "0.5")
                        :report "all"
                        :documentation "molar fractions in the tank")
                  ("xe" :is-a "variable"
                        :system-theoretic "input"
                        :indices ((:index "j" :lower "1" :upper "nc"))
                        :value ("0.2" "0.3" "0.5")
                        :documentation "molar fractions of the inlet"))
      :equations (("bal_ni" :is-a "equation"
                            :indices ((:index "j" :lower "1" :upper "nc-1"))
                            :relation ":diff<t> ni[j] == jen * xe[j] - jan * x[j]"
                            :documentation "component mass balances")
                  ("x_def" :is-a "equation"
                           :indices ((:index "j" :lower "1" :upper "nc"))
                           :relation "ni[j] == n * x[j]"
                           :documentation "definition of molar fractions")
                  ("sch_bed" :is-a "equation"
                             :relation "n == :sum<j,1,nc> ni[j]"
                             :documentation "summation condition for partial molar holdups")))
```

The variables `ni`, `x` und `xe` are specified as indexed variables. The option `:value` assigns an array of values to each of these variables. The component mass balances `bal_ni` and the definition equations for the molar fractions `x_def` are defined as indexed equations. In contrast the summation condition is a scalar equation.

6 Relations between modeling entities

In ProMoT three different relations between modeling entities exists.

- aggregations (containment),
- coupling relations,
- inheritance.

Aggregations and couplings are established while constructing larger modules bottom up out of smaller ones. Inheritance relations exist between sub- and superclasses. The different relations shall be presented here in depth, using the example with the cascade of two tanks (module `two-tanks` in Section 4.1.2).

6.1 Aggregations

Aggregations are vertical part-whole-relationships [AFGP96], which form a composed entity out of a number of other modeling entities. The composed modeling entity is called a container, the aggregated entities are the parts. Every part has an aggregation relation to the container. Composed modeling entities may themselves further be aggregated to more complex modeling entities. The parts are stored as named attributes of their containers which allow to identify them. These attributes are called slots.

Aggregation relations are allowed between the following pairs of containers and parts:

container	**part**
module	module
module	terminal
module	variable
module	equation
module	place
module	transition
terminal	variable
equation	variable

6.1.1 Strong and weak aggregations

ProMoT distinguishes two types of aggregations:

- strong aggregations and
- weak aggregations.

In a strong aggregation the part is dependent on the container. If the container is removed from the model, the part is necessarily also removed with it.

- The container is called strong Container or parent of the part.
- The part is called a strong part of the container.
- The named attribute or slot that contains the part inside the container is called a strong slot .

A strong aggregation is a 1:n relation, because a part has exactly one strong container and the container may have an arbitrary number of strong parts. For example the module `two-tanks` is the strong container of two modules of the type `tank`. This modules are contained in the strong slots `t1` and `t2`.

One important issue with strong aggregation is, that recursive aggregation of a module class in itself is not possible. Considering the example in Figure 10 it is an error to define the class `tank` with a strong part `t1` that is itself a `tank`. This would lead to a endless recursive loop and is not allowed[9].

In contrast in a weak aggregation, the part is not dependent on the container. The container may be removed from the model without affecting the part.

- The container is called a weak container of this part.
- The part is a weak part of the container.
- The named attribute or slot that contains the part is called a weak slot.

A weak aggregation is a n:m relation, since a part may have an arbitrary number of weak containers as well as a container may have many weak parts. In the example the terminals `in` and `out` are weak parts of the `two-tanks`, while they are strong parts of `t1` or `t2` respectively. In many cases interface variables like `jen` are strong parts of a module (`tank`) while they are weak parts of a terminal (e.g. `in` in the `tank`).

A strong slot is defined by the option `:is-a`, the argument to this option is the name of a modeling entity. This is called a instantiation of a modeling class. The strong slot contains the instance of the class. If no `:is-a` option is given and the part is not already defined in a superclass then as default the base class of the slot list is assumed as class for the slot. For instance in the definition of the `tank` the mass balance can also created by this equivalent definition:

```
49      ...
50      :equations (("bal_n" :relation ":diff<t> n == jen + jan"
51                           :documentation "mass balance")
52                  ...)
53      ...
54
```

Since in the section `:equations` only equations can be defined, and `bal_n` is not inherited from a superclass (this is called a direct part), it becomes an instance of the class `equation` implicitly.

Weak slots are defined by the option `:is-eq-to`, that has a slot-path as argument. This definition does not create a new instance or a copy of the original part, this slot refers to. It only establishes a reference to the part named by the slot-path and stores this reference under the given name in the container.

Slot paths: A slot path is a list of slot names, indexed slot names or back pointers (`"parent"`), that are separated with a dot (`"."`). With slot paths in ProMoT it is possible to access parts across one level of the aggregation hierarchy. Slot paths are mainly used to define weak aggregations and links. But they may also be used in equations and help variable values to access variables in terminals or submodules directly (see page 30).

[9] It is also impossible indirectly, so a module can never be a direct or indirect strong part of itself.

6.1.2 Internal part classes

A modeling entity can be instantiated many times as a strong part of other modeling entities. For example the module `tank` is used twice as a part of the cascade `two-tanks`. But both of these parts are independent of each other. And the `tank` can also be used alone. In its definition with `define-module` nothing is mentioned about its aggregation relations. And the `tank` itself is not changed with the definition of instances inside `two-tanks`.

For some strong slot definitions the instantiation is not that simple. The terminal `in` of `tank` is an instance of `terminal`. But it owns an additional slot `jn`, that is not present in `terminal`:

```
 7    ...
 8      :terminals (("in" :is-a "terminal"
 9                        :variables (("jn" :is-eq-to "parent.jen")))
10                  ("out" :is-a "terminal"
11                         :variables (("jn" :is-eq-to "parent.jan"))))
12    ...
13
```

ProMoT realizes this behavior by creating internal part classes. The modeler has no direct access to these internal classes. But the understanding of these internal structures may be helpful to use all features of the modeling system. The internal part classes shall be discussed

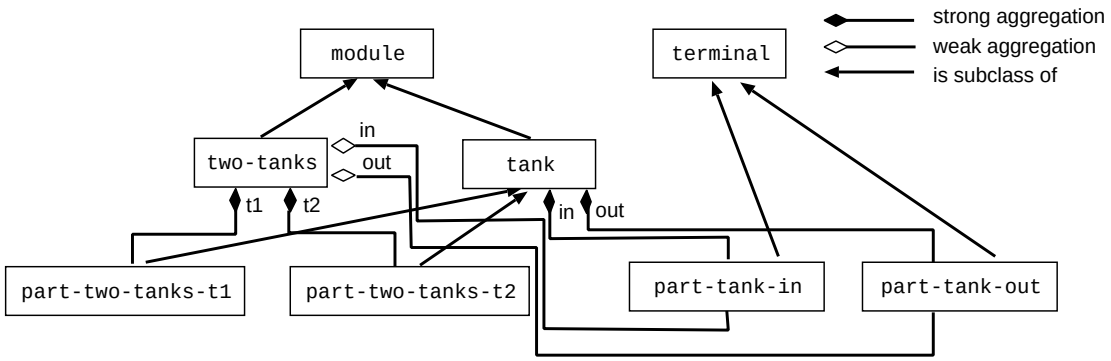


Figure 9: Aggregation relations and internal part classes in the construction of `two-tanks` out of `tank`

shortly with the example `two-tanks` from the Section 4.1.2. While defining the strong slot `t1` in the `two-tanks` ProMoT works as follows:

1. It creates an internal part class `part-two-tanks-t1` as direct subclass of `tank`. This class inherits all the attributes of `tank` and can be further specialized (e.g. with additional parts) in the instantiation.

2. It creates the slot `t1` and assigns it an instance of the part class.

Also in the definition of `tank` in Section 1.3.2 internal classes are created for the inflow and outflow (`part-tank-in` and `part-tank-out`). Figure 9 shows the internal classes and the different aggregation and inheritance relations.

6.2 Coupling relations

Couplings are horizontal part-whole-relations [AFGP96], that are created by ProMoT when couplings of terminals are created with the option `:links`. A link is a 1:n relation between terminals. There are some compatibility conditions that must be fulfilled for all terminals of a link:

- The terminals, that are connected in this link must not appear in other links.
- Every terminal must contain the same set of variables. The variables inside the terminal are identified by name, not by the position in the terminal.
- For each pair of connected variables the following compatibility conditions are checked:
  - The vectorial types must be compatible: potential variables must not be mixed with fluxes in couplings, i.e. it is not allowed to couple a potential `t1.p` with a flux `t2.p` in another terminal, although it is allowed to couple a potential `t1.p` with another potential `t2.p` and in the same link a flux `t1.f` with another flux `t2.f`.
  - all variables in one coupling must be scalar or the array dimensions must match. For indexed variables ProMoT creates indexed coupling equations automatically, that run across the whole range of the indices.

When these conditions are fulfilled, ProMoT creates coupling equations for the link. This equations are created for sets of vectorial variables (fluxes) and for non-vectorial variables (potentials) in different ways:

Fluxes: fluxes with the same name in a link sum up to zero, if the vectorial direction is considered. For the terminals $t_i, 1 \leq i \leq nt$ with the flux variable f the following equation is created: $0 = \sum_{i=1}^{n} tt_i.f$.

Potentials: potentials with the same name in a link are equal. For the terminals $t_i, 1 \leq i \leq nt$ with the potential variable p the following $nt - 1$ equations e_i are created: $e_i : t_1.p = t_{i+1}.p, 1 \leq i \leq nt$.

6.3 Inheritance relations

Inheritance relations are established in the definition of modeling entities as subclasses of other modeling entities as well as in the creation of strong parts (as explained in Section 6.1.2). A modeling entity inherits all

- strong and weak parts,
- property attributes and
- coupling relations

of its direct superclasses. The modeling entity may define additional, direct attributes and couplings. The set of all parts, properties and couplings of a modeling entity is defined as the union of the inherited and direct parts, properties and couplings respectively. Inherited parts can be specialized locally. For instance it is possible to define the value of a variable part in a module. It is also possible to change the type of an inherited module to a subclass of its original

type. It is not possible to forget or delete parts. Subclasses are in every case more specialized and should have more parts and attributes. They can not be more general than their superclasses and can not have less parts.

Multiple inheritance imposes some rules for the superclasses and the priorities of the inherited attributes. A class may have multiple direct superclasses, but these classes must not have an inheritance relationship with each other. Also the inheritance graph must form a directed acyclic graph, i.e. the graph must not contain cycles. A class may not be a direct or indirect superclass of itself. For discussion the example in Figure 10 is introduced.

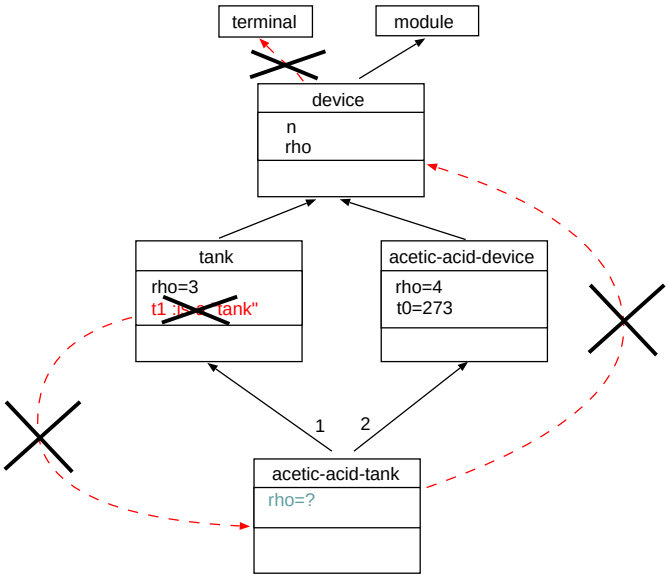


Figure 10: Examples for multiple inheritance and inheritance priorization.

The classes `tank` and `acetic-acid-device` are direct subclasses of `device`. Both inherit the variables `n` and `rho` from their superclass. The class `acetic-acid-tank` has both `tank` and `acetic-acid-device` as direct superclasses. It may not have device as direct superclass (right crossed out arrow) since `tank` is already a specialization of a `device`. Also `tank` may not have `acetic-acid-tank` as a direct or indirect superclass because this would form a cycle in the inheritance graph, which is not allowed. The sets of inherited attributes of the superclasses are unified, so `acetic-acid-tank` has `n`, `rho` as well as `t0`. Normally multiple superclasses should complement each other, so there should not be different definitions of the same attributes. In this example `tank` and `acetic-acid-device` both define the value of `rho`, but differently. This causes a conflict that is resolved by ProMoT using the order in the list of `:super-classes`. The definition of `acetic-acid-tank` defines

```
:super-classes ("tank" "acetic-acid-device")
```

so `tank` appears first in the superclass list and therefore its definition of `rho` is priorized. That is why the `rho` in `acetic-acid-tank` has a value of 3. This priorization rules apply not only for variables but for all inherited attributes and parts. For the crossed out relations in Figure 10 ProMoT issues error messages. Error messages are also issued if the definition of a modeling entity tries to merge two or more of the base classes supplied by ProMoT (see Figure 4) e.g.

tries to define

```
:super-classes ("module" "terminal")
```

The specialization hierarchies below each of the base classes must be disjunct.

It is possible to specialize inherited parts locally. It is also possible to change the kind of aggregation for an inherited part. The following pairs of inherited and local parts are allowed in ProMoT :

inherited part	local part
strong	strong
strong	weak
weak	weak

It is not allowed to override an inherited weak part with a local strong part (see also [Trä98]). In any case the local part must match the type of the inherited part. A inherited part can only be replaced by an instance of the same class or a subclass of it. Therefore it is not possible to replace the variable `c` that `:is-a "concentration"` by a variable that `:is-a "molar-flux"`.

7 The Graphical User Interface

The interactive modeling environment of ProMoT, version v-0-8-4 , comes with a **G**raphical **U**ser **I**nterface (Gui) that is programmed using the Java Foundation Classes (so called Swing). It is started automatically, when ProMoT is started from Emacs. An alternative to Emacs is to start ProMoT on a normal command line by the command `promot`. On the command line as well as in Emacs the two parts of the interactive modeling environment, the ProMoT kernel and the ProMoT Gui, are then launched as separate processes that interact with each other (see Figure 1). The Gui presents graphical views on the modeling-entities that are present in the kernel. It is possible to stop and restart the Gui (from the Emacs menu Promot > Start Promot User-Interface or with the command `promot-ui` on the command-line) without loosing data, since all the modeling data is contained in the kernel. This may be useful, if the Gui suddenly crashes. On startup the kernel and the Gui start and connect to each other. It is only possible for a user to have one ProMoT kernel running at the time, because this program registers with a global name and the Gui connects to this kernel. However, the user can have multiple Gui communicating with a single ProMoT kernel.

The Gui consists of several parts namely:

- The ProMoT Browser as the central unit to show the available modeling entities, to manage and find them and to start any of the other parts of the Gui.
- Some views that visualize the inheritance and aggregation relations of some selected modeling entities. These are the heritage views (subclass and superclass views) that show the ascendents and descendents of a modeling entity as a graph (instead of a tree as in the browser, which is not well suited for multiple inheritance). Further there is the aggregation view that presents the strong and weak aggregation parts (see Section 6.1) of one modeling entity. And finally there is a source view for showing the source code of the modeling entities and especially the variables and equations.
- The ProMoT Visual Editor allows to edit the structural aggregation of a single module. The user can change superclasses, submodules, terminals and links as in a drawing program.
- The ProMoT Visual Explorer is a flexible view and shows the aggregation of the structural entities in a module, i.e. submodules, terminals and links, across multiple levels of hierarchy as they where defined graphically in the visual editor.
- It is also possible to edit the source code of a modeling entity within the Gui. Therefore a rather rudimentary Java text editor can be used or the fully featured editor Emacs is invoked. Users are strongly encouraged to use Emacs, because it helps with features like syntax highlighting and parenthesis checking to find or avoid many typical errors.
- Last but not least the model must be written to Diva for simulation. Therefore the model output dialog is used. It allows to specify the filenames for the Diva model to generate and shows eventual error messages , if the model fails in consistency checks of ProMoT or the CodeGenerator.

All these parts and the editing possibilities are introduced in more detail in the following sections. Most of the interactions, the user can start in the Gui are initiated with the mouse. The left button selects modeling entities, menu entries or other graphical objects, the right button usually

presents a context menu for the current object. In the ProMoT Visual Editor many actions are performed using Drag&Drop functionality[10] to draw structural flowcharts.

7.1 ProMoT Browser

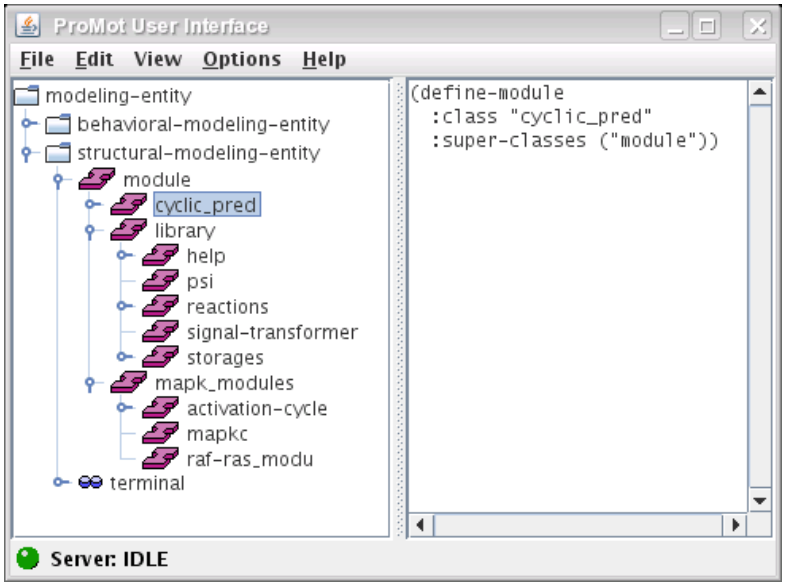


Figure 11: The class browser of ProMoT, showing example modeling entities for modeling in the field of Systems Biology.

The browser is the part of the Gui that is shown directly after the startup. It contains a tree view of the modeling entities that are currently present in the kernel and has a normal drop down menu. All other parts of the Gui can be started through interactions with the browser. The tree of the browser presents (as shown in Figure 11) the inheritance or specialization hierarchy of the modeling entities (see Section 6.3). ProMoT permits the use of multiple inheritance but the browser is only capable to present a tree with one ascendent per entry, which means that modeling entities, that have more than one superclass appear multiple times in the browser below of every of their superclasses. Since this may lead to confusion for difficult inheritance structures there are the heritage views which are able to illustrate multiple inheritance much nicer in a directed acyclic graph.

The basic modeling classes that are present after every startup of the browser are

- `modeling-entity`,
- `behavioral-modeling-entity`,
- `continuous-modeling-entity`,
- `equation`,

[10] Pressing the left mouse button on one object, holding the button and moving the regarding object to another place where the mouse button is released to put the object there.

- `variable`,
- `discrete-modeling-entity`,
- `place`,
- `transition`,
- `structural-modeling-entity`,
- `module` and
- `terminal`.

All interactions with the browser are performed with the mouse by clicking on the tree, the drop-down menu on top of the browser window or the pop-up menu that appears when clicking with the right button on one of the modeling classes in the tree.

7.1.1 Browser Tree

The browser tree shows the actual class library. If new classes are created or classes are deleted, it should reflect these changes. The tree entries show a special icon on the left which indicates if they have subclasses (e.g., class `cyclic_pred` in Figure 11). Clicking on this icon expands the view on the subclasses which has been done for `mapk_modules` in Figure 11. For using structural modeling entities the user can drag entities from the tree to the ProMoT Visual Editor. The tree shows also the documentation string for the classes under the mouse pointer in a tool tip (bubble help), if the mouse rests for some seconds on one specific entry. Information about the class is loaded from the ProMoT kernel into the tree as needed[11].

7.1.2 Pull-down Menus

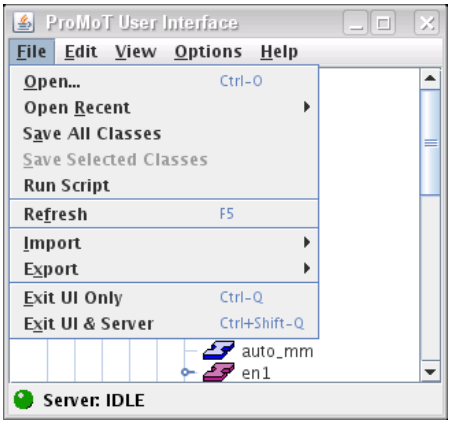


Figure 12: Pull-down menu of the browser. The 'File' menu was selected.

[11] This means, if a lot of classes have to be shown for the first time in the tree, there may be a delay. This applies especially if the Select menu is used.

The different pull-down menus of the browser are explained in the following. They are located above the tree view (see Figure 12):

File allows to load and save MDL files, to import and export files, to run scripts and to terminate PROMOT.

- **Open** Shows a file browser that allows to select MDL files (with suffix `.mdl`). This files are loaded into the PROMOT kernel after selection. In this way modeling libraries and predefined modeling entities can be included into the current modeling session. For modeling libraries it is possible to write loader files which control the input of all the other necessary files with the commands `cd` and `include` (see Section 3), so that only one file has to be selected for the library. If there are syntactical errors in these files, the wrong definitions (and only them) are not loaded and an error message is displayed.
- **Open Recent** Loads recently opened MDL files.
- **Save All Classes** Shows a File Browser to select a File or a new file name to save the complete library of modeling entities that is currently present in the kernel. The MDL source code of all these classes is then saved to one file, sorted in an order that a class appears in the file after all other classes it depends on (as `:super-classes` or by using them as parts with `:is-a`).
- **Save Selected Classes** Does the same as 'Save All Classes' but saves only the classes that are currently selected in the tree. You can manipulate the selection manually or by using the different functions of the menu Select (see below).
- **Run Script** Runs PERL scripts to generate MDL code.
- **Refresh** Updates the class tree.
- **Import** Imports different formats through several submenus. Supported are SBML models (up to Level 2 Version 1) and logical models (CELLNETANALYZER).
- **Export** Exports different formats through submenus. Supported are SBML models (up to Level 2 Version 1) and logical models (CELLNETANALYZER). PROMOT also writes optimized model code for DIVA, Diana and MATLAB.
- **Exit UI only** Terminates the GUI but not the PROMOT kernel it is connected to. All the data in the kernel stays intact.
- **Exit UI & Server** Terminates the whole PROMOT. Changed and unsaved modeling entities in the kernel are lost.

Edit This menu provides Undo/Redo functionality. It also starts editing functions for (multiple) selected classes in the tree. If there are no applicable classes selected, the functions are grayed. Some of the editing functions for the selected classes are also available in the pop-up menu of the browser tree.

- **Add Subclass** Adds a subclass to the selected class.
- **Copy Class** Creates a copy of the selected class. Afterwards rename the new class.
- **Edit Source** Starts a text editor for the source code of the selected classes, see the Edit Source (7.1.3) entry for the pop-up menu below.
- **Visual Edit Class** Launches the PROMOT VISUAL EDITOR for the selected class.
- **Delete** Deletes all modeling entities selected in the tree. All dependencies are also removed. Changes are lost.

Delete All Deletes all modeling entities loaded into the server. Changes are lost

Select It is possible to select classes in the browsers tree by direct interaction with the mouse. A single click with the left button selects one class. When holding the <Shift> key pressed on the keyboard while clicking, all visible class rows between the existing selection and the newly clicked class become selected. Another possibility is to use the <Control> key while selecting classes. Clicking on a unselected class while holding <Control> adds this class to the set of selected classes, whereas clicking on a selected class with <Control> removes this class from the selection. There are some automatic selection methods:

Select All Selects all classes currently present in the ProMoT system.

Select Subclasses Selects all subclasses of the latest selection in the tree. The tree is expanded as well to show all appearances of these classes.

Select Superclasses Selects all superclasses of the latest selection in the tree. The tree is expanded as well to show all appearances of these classes.

Select All Dependencies Select all dependencies of the currently selected class.

Search Search for a class using a small dialog. It expects the input of a full class name or a substring. When pressing the search button, all appearances of the class are selected and are highlighted in the tree.

Parametrize from Diva Parametrizes a copy of the selected class with data coming from Diva.

View The contents and relationships of a model in ProMoT can be explored by different views. Most of the views can be called from this menu.

View Source Starts a text viewer for the source code of the selected classes.

Visual Explorer Launches the ProMoT Visual Explorer for the currently selected class. The hierarchical structure of the model is shown to an arbitrary depth (recursive view).

Subclasses Launches a 'Heritage Viewer' which shows all subclasses of the current class.

Superclasses Launches a 'Heritage Viewer' which shows all superclasses of the current class.

Browse Diva Model Launches a model browser which shows all parts (terminals, variables and equations and also weak parts (references) of the selected module, including inherited parts.

Show Quick View If this is selected, the browser shows in its right part a small window with the source code of the currently selected class.

Options This menu shows user preferences and optional features of the Gui.

Mdl Options Provides options for the managing of Mdl code.

Diva Options Defining options for the Diva code.

Customize Undo/Redo Provide options for undo and redo functionality.

Use Emacs as Editor When this check box is available (not grayed out) and selected the Emacs is used as the text editor for editing source code of classes. Otherwise the very simple Java editor is used instead.

Help This menu gives options to further help information.

About Shows a 'About' dialog with information about curretn version of ProMoT and used external software packages and their licences.

Online Help Starts the default external web browser showing the homepage of ProMoT with further information.

7.1.3 Pop-up Menu

For every modeling entity in the tree of the ProMoT Browser a context sensitive pop-up menu can be shown with the right mouse button. The menu works for the "current modeling class" which is the class where the mouse was over when the right mouse button was pressed. In Figure 13 the class cyclic_prot_en1 is the current class.

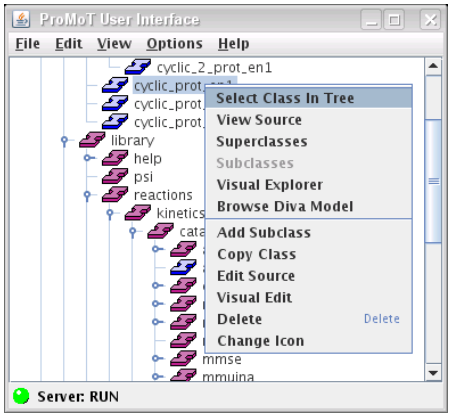


Figure 13: Pop-up Menu of the browser for the class cyclic_prot_en1. The item 'Subclasses' is grayed because this class has no subclasses.

Select Class In Tree Select the class in the browser tree.

View Source Launches a separate window that shows the source code of the current class.

Superclasses Launches a 'Heritage Viewer' which shows all superclasses of the current class.

Subclasses Launches a 'Heritage Viewer' which shows all subclasses of the current class.

Visual Explorer Launches the ProMoT Visual Explorer for the current class. This works for module classes only.

Browse Diva Model Launches the model browser which shows all parts (terminals, variables and equations and also weak parts (references) of the selected module, including inherited parts.

Add Subclass Asks for the name of a new subclass and then adds this class as an empty subclass of the current class. This is meant for creating a new modeling entity.

Copy Class Creates a copy of the selected class. Afterwards rename the new class.

Edit Source Launches a text editor for editing the source code of the current modeling class. The editor can be the JAVA text editor or an editing window of the EMACS if the regarding option in the 'Options' menu is active (see above).

Visual Edit Launches the PROMOT VISUAL EDITOR for the current class. This works for module classes only.

Delete Deletes only the current class from the running PROMOT system.

Change Icon Shows a file browser where a new icon file for the selected class can be choosen.

7.2 Heritage Views

Heritage Views show a directed acyclic graph (DAG) with all subclasses or superclasses of one modeling entity (see Figure 14) respectively. The top level superclasses or the selected superclass

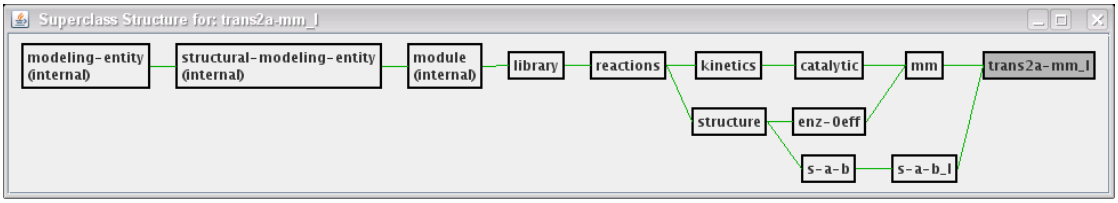


Figure 14: Superclass view for the class `trans2a-mm_l`. All different superclasses are shown with their different branches.

are always on the left. The selected subclass or the leaf node classes are shown at the right of the window. Leaf node classes like `trans2a-mm_l` in Figure 14 can be identified by their dark gray box. The PROMOT internal classes, that are not defined by the user, are marked with `(internal)`. The heritage view is updated, if the inheritance structure changes. For every class in the heritage view the same pop-up menu as in the PROMOT BROWSER (see Section 7.1.3) can be activated with the right mouse button.

7.3 Aggregation View

The aggregation view shows also a DAG and looks somehow similar to the heritage view. But it presents all the parts of the current modeling entity.

As Figure 15 shows, the top level container is presented on the left and all the parts follow in the graph to the right. It shows all parts including the direct parts defined by this modeling entity itself and the inherited parts defined in all of its superclasses. The graph of the strong parts forms a straight tree where all the parts on one level of the aggregation hierarchy are ordered in one column. The weak parts form somehow cross links in this hierarchy and are highlighted with green arrows. With the options dialog (see right part of Figure 15) that pops up for the menu View>Options, the parts to show can be selected. E.g. to hide all the variables, the user would deselect the check box for 'Variables' and press 'Apply'. Since the aggregation graph can easily exceed the limits of the screen for larger models, it is intended only for the use with small modules where the aggregation and inheritance of the behavioral parts is of interest.

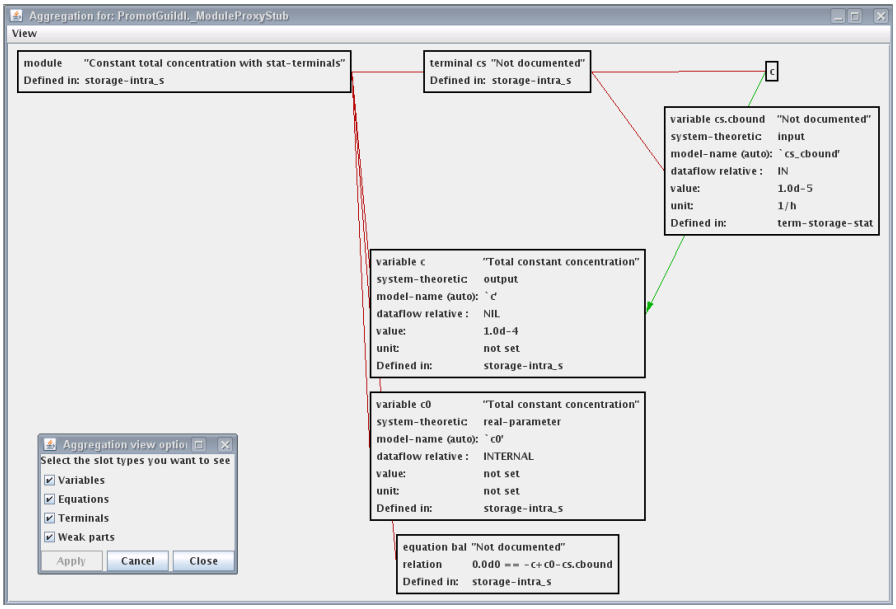


Figure 15: Aggregation view for `storage-intra_s`: The module contains a terminal `cs`, two variables and an equation. The terminal is the strong container for another variable `cbound` and uses the variable `c` as a weak part.

7.4 ProMoT Visual Editor

For definition of modules based on aggregation and coupling of other modules the visual editor can be used. It is launched by the pop-up menu of the ProMoT Browser, by double-clicking on a specific class in the tree or by the Heritage Viewer. The ProMoT Visual Editor works like a Drag&Drop-based drawing program for one module.

As Figure 16 shows, the editor window consists of two parts, the palette and the work pane. In the palette the modules and terminals can be selected in order to aggregate them to the module. Modeling entities in the palette are drawn in the palette with their name and their graphical representation, which is also used in the work pane on the right hand side. Under the tabs 'Terminals' and 'Modules', terminal and module classes can be found respectively. It is possible to put new classes into the palette via Drag&Drop from the ProMoT Browser. In the tab 'Superclasses' the current superclasses of the module are shown. It is also possible to change the superclasses by dragging other module classes into this pane. But it is possible that a new superclass replaces an old one completely to fulfill the inheritance rules (see Section 6.3). Classes can be deleted from all three palette panes with the context menu of the right mouse button. If an existing non-empty module is opened into the ProMoT Visual Editor, the palette contains all the modeling entities that are already used as part of the module.

The right work pane contains a window (work window) where parts from the palette can be dropped on. Submodules are usually drawn as rectangles anywhere in the window and external terminals are rectangles that stick to the border of the window. Links are represented in the drawing as lines connecting the terminals of submodules. If an external terminal is connected to a terminal of a submodule, than this means a weak aggregation of the terminal of the submodule as terminal of the currently edited module. In Figure 16 e.g., `t_eiia` (on top) is a terminal that

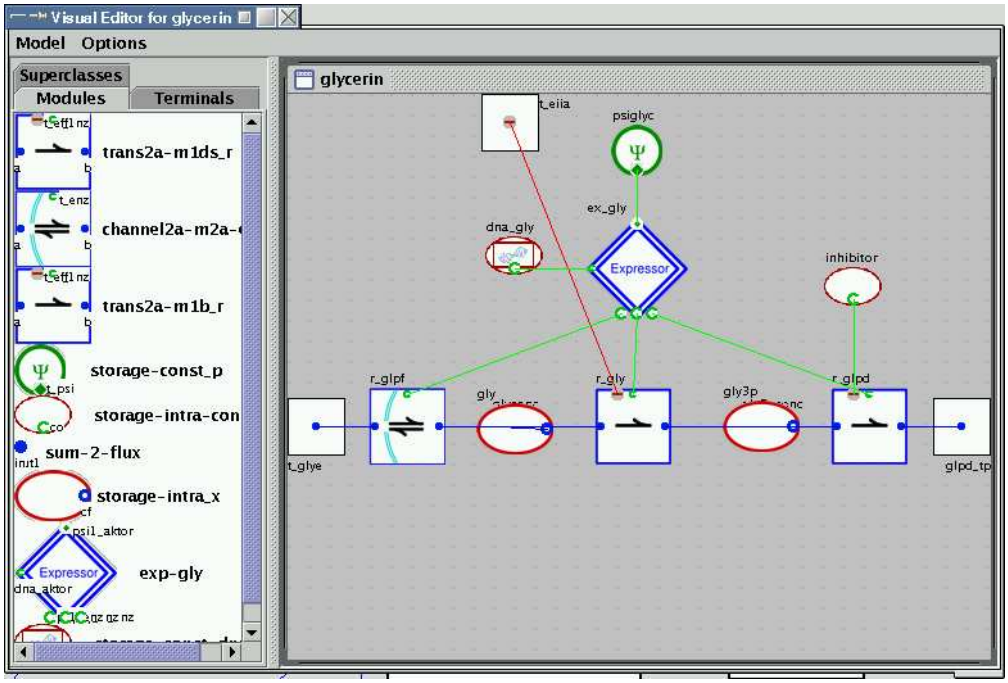


Figure 16: Visual editor for a cellular transporter module.

`:is-eq-to` a terminal of the submodule `r_gly` and `gly` is a module. A basic feature of this work pane is to select one or more of the shown parts. This is done by clicking with the left mouse button on the part. Multiple parts can also be selected (e.g. for positioning) by specifying a selection rectangle with the left mouse button or by clicking with the mouse while holding the <`shift`> key on the keyboard.

The parts in the work pane are drawn using icons, if there are icons defined for the classes of the modeling entities (`:icon`). Normal colored `gif` raster images can be used as icons, but the user has to provide them by using external tools (e.g. Gimp, Xfig etc.). It is possible to paint modules and terminals in other shapes than rectangles (as shown for the submodule `gly`) in the drawing by defining transparent areas in the icons. If an icon is specified, the size of the raster image defines also the size of the submodule in the drawing. Otherwise if a submodule has no icon the size is set to $\frac{1}{5}$ of the size of this module ($scalefactor = 0.2$) in the ProMoT Visual Editor. This can be changed by opening the module in the ProMoT Visual Editor and changing the size of the work window. The size of a module in screen pixels is stored in Mdl in the attributes `:geometry-width` and `:geometry-height` . If no size is defined for a module, the default size is $[300 \times 300]$ pixels. Submodules in the work window can be positioned by selecting them with a click of the left mouse button and dragging them around. The position of the center of a submodule in pixels is stored in the Mdl attributes `:geometry-x` and `:geometry-y`. If no position is given, the default is $[0, 0]$. The terminals of a submodule (inner terminals) are presented as small squares along the edges of the shape of the submodule, or with their respective icons. Their position is fixed and depends on the definition of the class of this module instance. The names of this inner terminals can be displayed optionally. Because this leads sometimes to a cluttered display, the user can disable this in the options menu of the ProMoT Visual Editor

(Options > Show Terminal Names). The external terminals have a position defined relative to one edge of the module. The respective edge is stored in MDL with the `:geometry-side` which can be one of `"top"`, `"bottom"`, `"left"` and `"right"`. The position (`:geometry-position`) is defined as the relative distance from the top (on vertical edge) or left (on horizontal edge) corner of the module in a float value in $[0.0, 1.0]$. The default values are `"left"`, `"0.5"`. Also external terminals can be positioned dragging them with the mouse along the edges of the module. If the module is resized, they keep their relative positions. This positions are also used to draw the terminals, when the module is used as a submodule.

Links are lines between terminals of the submodules. They can be drawn by pressing the left mouse button on a free terminal, holding the button and dragging the appearing rubber band line to another terminal. The link appears, if the terminals to connect are compatible (see Section 6.2). The link lines may have specified colors. This colors can be defined for the link or one of the connected terminals by `:link-color` in the source code. The value of this attribute can be a color name of `"black"`, `"blue"`, `"cyan"`, `"gray"`, `"green"`, `"magenta"`, `"orange"`, `"pink"`, `"red"`, `"white"` or `"yellow"`. If two or more terminals in a link specify different colors, the color of the first terminal encountered in the link, is used. The link color is an attribute should be preferably specified for a predefined terminal class to show the kind of connection that is established by the link.

A strong terminal of the module can be created by dragging a terminal from the terminal palette into the work window. Weak terminals can be established in two ways:

- By creating an arbitrary strong terminal and connecting it (like drawing a link) with a free terminal of a submodule. The strong terminal is then automatically transformed into a weak terminal.
- Using the context menu of a submodule by clicking with the right mouse button on the submodule. In the submenu 'Connect Terminals' there is an entry for every terminal of the submodule to propagate it as an external terminal. After selecting one of the menus entries, a new external terminal appears, that is a weak part referring to the named terminal.

All parts in the edited module are named. When creating new parts, they get a unique name based on their class name automatically. The name is shown for submodules above their graphical representation and for terminals below or right of the graphical representation. The name can be edited by selecting an element with the left mouse button and pressing any alphanumeric or the <`backspace`> key. Then the name becomes editable and can be changed. The editing is finished with the <`return`> key. Alternatively the name of a part can also be changed by the 'Edit Details' dialog, which can be reached by the context menu.

For the geometric layout of the model, all parts can be moved by dragging them with the mouse. Several parts can be selected and moved together. Therefore a rectangular selection can be drawn by pressing the left mouse button on the free background of the work window, holding it down and dragging a rectangle across all parts that should be selected. Alternatively one can use the left mouse button together with the <`shift`> key to add or remove elements to the selection. Another method for editing the layout is the so called broom. When pressing the <`control`> key and the left mouse button at the same time and moving the mouse horizontally or vertically, a thick line appears that pushes all parts to a certain line. The length of the line can be extended by moving the mouse orthogonally to the brushing direction. The broom is also available by a button on top of the work window.

By pressing the right mouse button on a part in the work window, a context menu appears, that allows several editing options for that part.

View Class Source: The current part is a strong part of the edited module and the source code of the `:is-a` class can be opened in a text viewer.

Select Class in Tree: The class is selected and shown in the tree of the browser. This is introduced because the browser allows all kinds of editing actions on classes and it does not make sense to include all of them in the menu of the visual editor.

Edit Details: Opens a properties dialog that allows at the current state of implementation, to change the name of the part and, if it's a strong part, to change the `:is-a` (Slot Class) class of the part. The included list for the selection of the Slot Class contains all the matching classes from the palette, but it is not only possible to select a class, but alternatively the class name can also be typed into the text field. Editing of variables is also provided. By pressing the 'Set' button in the dialog, the edited attributes are changed in the ProMoT model.

Visual Edit Class: Opens a new ProMoT Visual Editor window for the `:is-a` class of the current strong part.

Delete: Removes the submodule or terminal from the current module. This is only possible, if the part is a direct part of the module and was therefore not inherited from a superclass.

Connect Terminals: Allows to propagate one of the internal terminals of the current submodule as a weak external terminal of the currently edited module.

Since modules in ProMoT are classes that may contain inherited parts , some non obvious rules apply for the editing. This may lead to some pitfalls:

- Inherited parts can not be renamed. If a submodule or terminal is originated in a superclass, the name is given in that class and can only be changed, when editing the superclass. Therefore all renaming possibilities are deactivated in the visual editor.
- Inherited parts can't be deleted. As explained in Section 6.3 a subclass can only define additional or more specific parts. All inherited parts are also present in the subclass and all deleting functions of the editor are therefore disabled.

7.5 ProMoT Visual Explorer

The ProMoT Visual Explorer shows the aggregation of the structural modeling entities with the geometrical diagram information and the icons similar to the visual editor. But here the presentation is not limited to one single level of the hierarchical structure but can be extended (as shown in Figure 17) to an arbitrary depth.

The submodules and the links between them are shown for the current module and also for all submodules. Since a complex model would not fit completely to the screen if all details are presented at once, the user has to select the module of interest. The ProMoT Visual Explorer offers several modi for exploration:

Hierarchy Zoom: By clicking into a certain module with the left mouse button, this module is zoomed to window size. Internal details becomes visible but other parts of the model get zoomed out of the window. For zooming out just click with the left mouse button somewhere outside of the module.

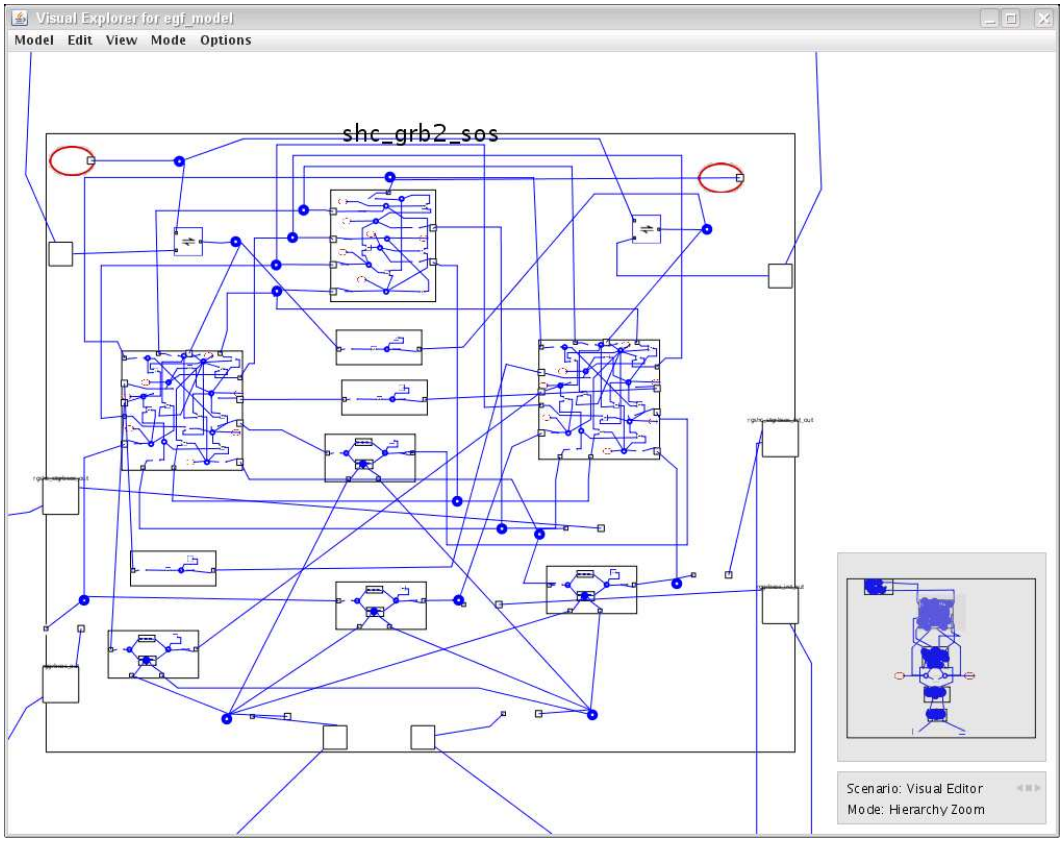


Figure 17: Recursive view of the module of the EGF signal transduction model. Different layers provide additional information, e.g. overview and navigation layer (on the bottom right).

Fisheye Zoom: By clicking the focused module is enlarged whereas all other modules are shrinked but are not hidden nor zoomed out of the window. In this way, the context of the focused module is preserved.

Interactive Zoom: This navigation mode is comparable to the Pan&Zoom functionality provided in other tools. Pan means moving the network in two dimensions. Zoom changes the scale at which the network is viewed (enlarge modules of interest or shrink them).

Focus Zoom: The focused module is positioned fully into the view but with minimal pan of the network. Hence, the user is not confused by too many Pan&Zoom steps.

The ProMoT Visual Explorer is not only used for exploring of the model but also for creating and manipulating different graphical representations of a model. Furthermore, the visual explorer is appealing for publication, exportation, and printing.
For an overview of navigation pattern, shortcuts and modifiers see appendix F.

7.6 Text Editor

Figure 18: Emacs for editing Mdl source code with error message.

For changes of the source code of a modeling entity a text editor is employed. Depending on the settings in the 'Options' menu (see page 55) either a Java editor or an Emacs window pops up. The Java editor is very simple, after changing the text, the window can be closed and the text gets interpreted as a new class definition. If there are errors in the text, a window with the error messages pops up. For more sophisticated editing the use of Emacs is recommended. Emacs

shows the source code with syntax highlighting and also displays the matching parenthesis in the code. After finishing the changes, the buffer should be saved and then be killed by the Menu Files>Kill Current Buffer. EMACS will ask, if the command should be evaluated in PROMOT, users can choose "no" to discard their changes here. If the command gets evaluated and errors occur, these errors get presented in a new EMACS buffer for correcting the source code. All syntax errors are presented with the line number of the wrong statement. By hitting the <`return`> key on the error message, the EMACS jumps to the respective line in the source code (see Figure 18).

7.7 Model Output Dialog (for Diva)

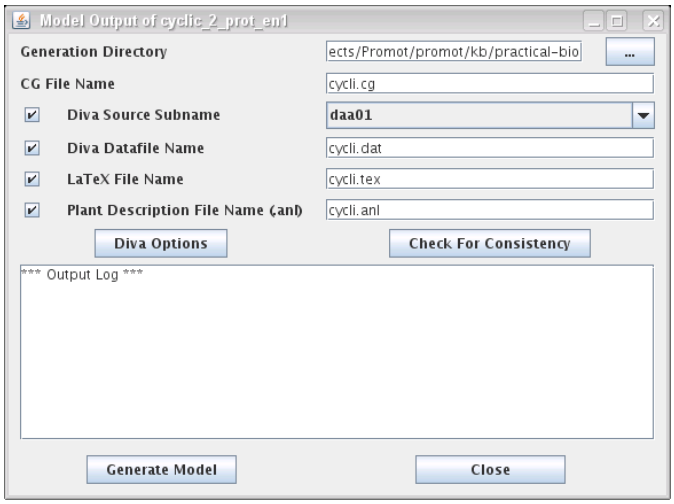


Figure 19: The Model Output Dialog for writing a model to DIVA.

With the model output dialog (shown in Figure 19) a module is written as a DIVA model. Hereby, the module is instantiated, and checked for completeness and consistency, e.g. if all `:values` for the variables are defined and all internal terminals are connected. If a model is written for the first time, it is very likely to get some error messages at this point of the development process. Then the equation system may be optimized, if the user selects this option. Before handing over the model to the CODEGENERATOR, the consistency of the equation system is checked, i.e. if the number of state variables matches the number of equations matches in the model. The model names of all the variables are then generated according to the rules, DIVA imposes for the different system-theoretic types. Here the `:model-name` (see page 17) is used, if it was specified for a modeling entity in the model. Then a CODEGENERATOR input file is written and the CODEGENERATOR is called to translate the model to DIVA code. The CODEGENERATOR may write out some messages during this process.

The model output dialog for DIVA has text fields to specify all the different file names for this process. If the MDL code of the module already contains this information (see Section 4.1.4), that values are used here as defaults. The 'Generation Directory' is used to write all the different output files in this process and is typically the directory that is used for all the simulation data in a model. The CG File name is the name of the generated input file of the CODEGENERATOR. This file is useful to look for the generated model names. It contains the shortened form as well as

the full hierarchical name in the `:meaning` attributes of the CODEGENERATOR definitions. The following four text fields specify the generated files for the diva simulation. They are explained in Section 4.1.4. If one of these file names is deselected with the check box, it will be not generated. If one of the files is already present, it gets overwritten without warning.

Below the fields for setting up file names the 'Diva Options'-button appears. It opens an additional dialog where options for the DIVA code can be defined. The slider for the 'CG Verbosity' can be used to adjust the amount of messages the CODEGENERATOR produces while running. Usually it is set to 1, but for debugging it may be set to 3 to see, at which element exactly the CODEGENERATOR encounters problems. Into the 'Output Log' field below some progress information and the messages of the CODEGENERATOR are written successively. The button 'Generate Model' starts the output process. If this leads to no error messages and the last message in the output log is 'All done' then a new DIVA model was generated successfully and the user can continue compiling the model with `dmake` and the run simulations in `drun`. See the Diva User Manual [HKK+00] for further details about the simulation.

A Mdl-Command reference

A.1 Strings and Symbols

⟨alpha-character⟩ :== `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z` | `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w` | `x` | `y` | `z`

⟨numeric-character⟩ :== `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

⟨special-character⟩ :== `-` | `_`

⟨alphanumeric-character⟩ :== ⟨alpha-character⟩ | ⟨numeric-character⟩

⟨character⟩ :== ⟨alphanumeric-character⟩ | ⟨special-character⟩

⟨string⟩ :== {⟨character⟩}$^+$

⟨quoted-string⟩ :== `"` ⟨string⟩ `"`

⟨list-of-quoted-strings⟩ :== `(` {⟨quoted-string⟩}$^+$ `)`

A.2 Mathematic Expressions

The syntax of ⟨unquoted-math-expression⟩ in Mdl is in most cases the same as ⟨arex⟩ in the CodeGenerator input language [Kö00], only ⟨variable-ref⟩ is defined differently and the definition of ⟨diff-expr⟩ and ⟨math-relation⟩ is added.

⟨variable-ref⟩ :== ⟨slot-path-element⟩ | ⟨unquoted-slot-path⟩

⟨diff-expr⟩ :== ⟨unquoted-math-expression⟩ `* :diff<t>` ⟨variable-ref⟩

⟨math-relation-side⟩ :== ⟨unquoted-math-expression⟩ | ⟨diff-expr⟩ { `+` ⟨unquoted-math-expression⟩ | ⟨diff-

⟨unquoted-math-relation⟩ :== ⟨math-relation-side⟩ `==` ⟨math-relation-side⟩

⟨math-expression⟩ :== `"` ⟨unquoted-math-expression⟩ `"`

⟨math-relation⟩ :== `"` ⟨unquoted-math-relation⟩ `"`

⟨array-of-math-expressions⟩ :==`(` {⟨math-expression⟩}$^+$ | {⟨array-of-math-expressions⟩}$^+$ `)`

A.3 Slots und Slot Paths

⟨unquoted-slot-name⟩ ⟨alpha-character⟩ {⟨alphanumeric-character⟩ | `_` }*

⟨slot-name⟩ :== `"` ⟨unquoted-slot-name⟩ `"`

⟨list-of-indices⟩ :== `[` ⟨unquoted-math-expression⟩ { `,` ⟨unquoted-math-expression⟩ }* `]`

⟨slot-path-element⟩ :== ⟨unquoted-slot-name⟩ | ⟨unquoted-slot-name⟩ ⟨list-of-indices⟩

⟨unquoted-slot-path⟩ :== ⟨slot-path-element⟩ | `parent .` ⟨slot-path-element⟩

⟨slot-path⟩ :== `"` {unquoted-slot-path}$^+$ `"`

⟨list-of-slot-path⟩ :== `(` ⟨slot-path⟩ `)`

A.4 Names for modeling entities

⟨class-name⟩ :== ⟨quoted-string⟩

⟨list-of-class-names⟩ :== `(` {⟨class-name⟩}$^+$ `)`

A.5 Commands

⟨command⟩ :== ⟨definition-command⟩ | ⟨file-command⟩
⟨definition-command⟩ :== ⟨define-module⟩ | ⟨define-terminal⟩ | ⟨define-variable⟩ | ⟨define-equation⟩
⟨file-command⟩ :== ⟨include⟩ | ⟨cd⟩
⟨define-module⟩ :== `(define-module :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:modules (` {⟨module-slot⟩}$^+$ `)`]
[`:terminals (` {⟨terminal-slot⟩}$^+$ `)`]
[`:equations (` {⟨equation-slot⟩}$^+$ `)`]
[`:variables (` {⟨variable-slot⟩}$^+$ `)`]
[`:links (` {⟨link-slot⟩}$^+$ `)`]
`)`
⟨define-terminal⟩ :== `(define-terminal :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:geometry-side "top" |"bottom" |"left"|"right"`]
[`:geometry-position` {⟨property-slot⟩}$^+$]
[`:link-color` {⟨color-name⟩}$^+$]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:variables (` {⟨variable-slot⟩}$^+$ `)`]
`)`
⟨define-variable⟩ :== `(define-variable :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:system-theoretic "state" | "input" | "output" | "help" | "structure-parameter" |`
`"real-parameter" | "integer-parameter" | "index" ]`
[`:positive-vector-direction "in" | "out"`]
[`:value` ⟨math-expression⟩ | ⟨array-of-math-expressions⟩]
[`:minimum` ⟨math-expression⟩]
[`:maximum` ⟨math-expression⟩]
[`:relative-error`[12] ⟨math-expression⟩]
[`:absolute-error` ⟨math-expression⟩]
[`:unit` ⟨math-unit-expression⟩]
[`:report "lumped"|"distributed"|"all"|"none"`]
[`:variables (` {⟨variable-slot⟩}$^+$ `)`]
`)`
⟨define-equation⟩ :== `(define-equation :class` ⟨class-name⟩
`:super-classes` ⟨list-of-class-names⟩
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:relation` ⟨math-relation⟩]
[`:variables (` {⟨variable-slot⟩}$^+$ `)`]
`)`

⟨property-slot⟩ :== `(` ⟨slot-name⟩ `:value` ⟨quoted-string⟩
[`:one-of` ⟨list-of-quoted-strings⟩]
`)`

⟨module-slot⟩ :== `(` ⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:terminals (` {⟨weak-slot⟩}$^+$ `)`]
[`:variables (` {⟨weak-slot⟩}$^+$ `)`]
`)`

⟨terminal-slot⟩ :== `(` ⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:variables (` {⟨weak-slot⟩}$^+$ `)`]
`)` | ⟨weak-slot⟩

⟨variable-slot⟩ :== `(` ⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:system-theoretic "state"` | `"input"` | `"output"` | `"help"` | `"structure-parameter"` |
`"real-parameter"` | `"integer-parameter"` | `"index"`]
[`:positive-vector-direction "in"` | `"out"`]
[`:value` ⟨math-expression⟩ | ⟨array-of-math-expressions⟩]
[`:minimum` ⟨math-expression⟩]
[`:maximum` ⟨math-expression⟩]
[`:absolute-error` ⟨math-expression⟩]
[`:unit` ⟨math-unit-expression⟩]
[`:report "all"` |`"lumped"` |`"distributed"`]
[`:variables (` {⟨weak-slot⟩}$^+$ `)`]
`)`

⟨equation-slot⟩ :== `(` ⟨slot-name⟩ [`:is-a` ⟨class-name⟩]
[`:documentation` ⟨quoted-string⟩]
[`:properties (` {⟨property-slot⟩}$^+$ `)`]
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
[`:relation` ⟨math-relation⟩]
[`:variables (` {⟨weak-slot⟩}$^+$ `)`]
`)`

⟨link-slot⟩ :== `(` ⟨slot-name⟩ `:terminal1` ⟨slot-path⟩
`:terminal2` ⟨slot-path⟩
[`:indices (` {⟨index-definition⟩}$^+$ `)`]
`)`

⟨weak-slot⟩ :== `(` ⟨slot-name⟩ `:is-eq-to` ⟨slot-path⟩ `)`

⟨index-definition⟩ :== `(:index` ⟨slot-name⟩ `:lower` ⟨math-expression⟩ `:upper` ⟨math-expression⟩ `)`

⟨include⟩ :== `(include :file` ⟨unix-file-name⟩ `)`

⟨cd⟩ :== `(cd :dir` ⟨unix-directory-name⟩ `)`

⟨unix-file-name⟩ and ⟨unix-directory-name⟩ are not further specified here. Please use a unix documentation.

B Introduction to EBNF

For syntax descriptions in this documentation the following grammar notation (Extended Backus Naur Form) is used:

- Symbols in angles (⟨nonterminal⟩) describe non terminals that have to be replaced according to the rules of the grammar.
- Symbols in bold face (`terminal`) identify terminals (in the sense of the grammar) that must be used literally.
- A production rule in the grammar (⟨nonterminal⟩ :==right-hand-side) describes, in which way a non terminal may be replaced. On the right hand side of „:==“ follows a list of terminals or non terminals that can be inserted instead of the non terminal. For instance describes the grammar:

  ⟨A⟩ :== `a`
  ⟨A⟩ :== ⟨A⟩ `a` ⟨B⟩
  ⟨B⟩ :== `c b`

  among others the following strings:
  `a, cb, aacb, aacbacb ...` Starting with ⟨A⟩ or ⟨B⟩ the terminal strings `a` or `c b` can be derived. If rule 2 is used (possibly recursive), the string starting with `aacb` can expand in both directions with `a` and `acb` respectively. Note that a production with these rules is only finished, if all no terminals have been replaced with terminals.
- Different rules of the same non terminal can be compacted by combining the right hand sides of the rules with “|”. The example grammar above can be written with the following completely equivalent rules:

  ⟨A⟩ :== `a` | ⟨A⟩ `a` ⟨B⟩
  ⟨B⟩ :== `c b`

- If a sequence of terminals and non terminals can appear optionally, they are written in brackets[13] “[...]”.
- If a sequence of terminals and non terminals is written in braces with a following plus “$\{...\}^+$” or star “$\{...\}^*$” this means that this sequence can appear in this position repeatedly. The plus means that the enclosed symbols must appear at least one time and with the star the symbols can also appear zero times (this means an empty sequence of symbols). Attention please: „`{`“ denotes a terminal in the grammar that is used literally in MDL whereas „{“ is the just described metasymbol.
- Additionally to the rules of the grammar a special non-terminal (starting symbol) must be defined, which is the starting point for every derivation of symbols. If this is not defined explicitly, than the first non-terminal of the first rule of the grammar is assumed implicitly as the starting symbol.

[13] Attention: The brackets in BNF have also a meaning as a terminal symbol in the slot paths of MDL. “`[`” is the terminal to start the subscript of an indexed modeling entity (e. g. `a[i]`) and “[” marks the given meta symbol in EBNF

- **Examples:** The following short grammar describes all character strings that contain at least one "a" at the beginning and may contain at most one "b" at the end.

  ⟨S⟩ :== ⟨A⟩ | ⟨A⟩ `b`
  ⟨A⟩ :== `a` | ⟨A⟩ `a`

  The following two rules describe the same set of character strings.

  ⟨S⟩ :== {`a`}* `a` [`b`]
  ⟨S⟩ :== {`a`}+ [`b`]

C Getting Started

ProMoT can be used in two flavors: as interactive modeling environment `promot` or as command line compiler `mdl2diva` for finished models. The interactive environment is usually started from the text editor Emacs. Please follow the instructions to launch ProMoT for the first time. Prerequisite to these instructions is, that you already have an account on a Unix computer where ProMoT is installed and you are logged in.

1. **Starting Emacs:** Open a shell window like `xterm` or the `konsole`. On the command prompt in this window type `emacs &` <`return`>[14]. The result is, that an editor window pops up, which has a title bar that usually states something like "emacs@<hostname>" where <hostname> is the name of your computer.

2. **Extending the paths**

   (a) **Find out your shell** Type `echo $SHELL` into the shell window. This shows you the name of your shell program. Depending on this you have to select in which way you can change your paths. If the output of this command ends with `/sh` or `/bash` or `/ksh` then you have to change your `~/.profile`, which is described from point 2b. Otherwise if the program ends with `/csh` or `/tcsh` then you should change your `~/.cshrc` instead, which is described from point 2d.

   (b) **Open .profile**. Open the file " /.profile" in the Emacs by using the menu: Files>Open file. The Emacs expects the input of the filename in the so called minibuffer, the bottommost line of the window. Type in /.profile `return` there. The contents of a file is shown in the editor window, or an empty window, if the file has not been already there.

   (c) **Change .profile**. Move the cursor to the end of the file and add the following lines:

   ```
   PATH=${PATH}:/usr/local/diana/bin
   export PATH
   ```

   There should be a empty line at the end of the file!. Save the file by the menu: Files>Save Buffer. You can close the file in Emacs by the menu: Files>Kill Current Buffer. Continue with 2f.

   (d) **Open .cshrc** Open the file " /.cshrc" in the Emacs by using the menu: Files>Open file. The Emacs expects the input of the filename in the so called minibuffer, the bottommost line of the window. Type in /.profile `return` there. The contents of a file is shown in the editor window, or an empty window, if the file has not been already there.

   (e) **Change .cshrc**.Move the cursor to the end of the file and add the following lines:

   ```
   setenv PATH ${PATH}:/usr/local/diana/bin
   ```

   There should be a empty line at the end of the file!. Save the file by the menu: Files>Save Buffer. You can close the file in Emacs by the menu: Files>Kill Current Buffer. Continue with 2f

[14] The syntax <...> is used for single keys on your keyboard. In this case the "enter" or "return" key is meant. For the following commands the return key will not be mentioned explicitly. If the instruction in this document is "Type xyz to abc" then you should switch the input focus to the window abc, type xyz there and hit the <`return`> key.

(f) **Testing the paths** Log out from the computer and log in again. Open the shell window and the EMACS as described in 1. Type `which mdl2diva` into your shell window. If it writes out `/usr/local/diana/bin/mdl2diva`, then everything is OK and you can continue with 3. Otherwise double check, if you did the steps correctly and perhaps ask you local guru.

3. **Installation of ssh keys** Now we install some keys for the ssh, that should allow an easy remote access to other computers in the local area network, without typing passwords every time. PROMOT uses this feature sometimes, so we should ensure, that this works.

   (a) **Check for old keys:** Type `ls ~/.ssh` into your shell window. If the files:

   ```
   authorized_keys2
   id_dsa
   id_dsa.pub
   id_rsa
   id_rsa.pub
   ```

   are already present, then it is most likely that everything already works for you. Test it with 3c. If it doesn't work, continue with the installation from 3b

   (b) **Generating ssh keys** Type the following sequence of commands into your shell window:

   ```
   mkdir ~/.ssh
   ssh-keygen -b 2048 -f ~/.ssh/id_dsa -t dsa -N '' -q
   cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys2
   ```

   The first command `mkdir` may report `mkdir: cannot make directory '.ssh': File exists`. This is not a problem.

   (c) **Testing ssh:** When everything went OK up to this point, you should be able to log into every computer in the local area network without typing passwords. Type `ssh ssh` into your shell window. The program should not ask with

   ```
   Password for <user>@ssh:
   ```

   for your password.

   If this works, continue with 4. If something is wrong, please double check if you did the steps correctly and perhaps ask you local administrator.

4. **Installation in Emacs** For making PROMOT available in EMACS, you have to edit your personal EMACS startup file.

   (a) **Open .Emacs** Open the file `~/.emacs` in the EMACS by using the menu: Files>Open file.

   (b) **Add promot line** Insert the line:

   ```
   (load "/usr/local/promot/promot/emacs/promot")
   ```

   as the first line of your `~/.emacs`. It will be executed every time you start EMACS and makes it possible to use PROMOT within EMACS. Save the file using the menu: Files>Save Buffer. You can close the file in EMACS by the menu: Files>Kill Current Buffer.

(c) **Testing ProMoT in Emacs** Quit EMACS with Files>Exit Emacs and restart it, as shown in 1, from the shell window. It should show now a additional menu "Promot" on the left of the menu bar. Select Promot>Start Promot from the menu bar and see what happens. After some seconds there should be a emacs window `*promot-server*` in which a greeting message like:

```
; This is Promot v-0-8-1
; Copyright (c) 1997..2009
; Martin Ginkel, Frank Traenkle, Volker Padur, Sebastian
Mirschel, Katrin Kolczyk, Michael Rempel, Erik Aschenbrenner, Michael Krasnyk, Alexey
; MPI Magdeburg
;
; Image created on 2008-8-8 16.53:10 by rempel
```

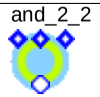
shows up. On the other hand the PROMOT GUI should start up, showing the PROMOT BROWSER window.

5. **You are now ready to model!**

D The basic elements of the library for logical modeling

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
module	yes	structural-modeling-entity		
struct-library	yes	module		
compound	yes	struct-library		Compounds are used to model logical states in a signaling network. The subclasses contain some common molecular classifications of certain species. Note that compounds can not be directly connected, and you need (at least) one element of the class gate between any two compounds (superclass for species in the network).
adapter	no	compound		
drug	no	compound		
dummy	no	compound		Dummy is a special class. It can be used to insert certain properties e.g. time delays in a model. Therefore dummies are useful in a mathematical sense but have no biological interpretation.
external-stimulus	no	compound		
gap	no	compound		
gef	no	compound		
gprotein	no	compound		
gtpase	no	compound		

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
kinase	no	compound		
ligand	no	compound		
other	no	compound		The class other should be used if the species type is not really known or not contained in the library as subclass of compound.
phenotype	no	compound		
phosphatase	no	compound		
receptor	no	compound		
reservoir	no	compound		reservoir is a special class. Elements of this class are used when several pools of a molecule are considered. This is a common situation, for example, if two subsets of a certain molecule with distinct properties, typically a different localization in the cell or different regulation, are to be modeled. In this case, a "reservoir" is included which is required for the activation of both pools. This allows simultaneous knock-outs of both pools, but this is only a mathematical entity without a clear biological interpretation. See also tutorial for logical modeling
transfactor	no	compound		
ubiquitin-ligase	no	compound		

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
gate	yes	struct-library		Gates connect different components of the network in a logical manner. They describe certain reactions and are based on common boolean logic and operators.
gate-2	yes	gate		Class defining structural properties for gates with two inputs.
activ	yes	gate		A single arc denoting the influence of one element on another is made via an activ (activating gate). If the effect is negative, a not should be additionally included.
and	yes	gate		The and gate allows the connection of two or more compounds together and follow the all or nothing normal logic of an and gate. We have built and gates that allow up to 6 input species, however larger gates can also be built.
and-2	yes	and, gate-2		The class `and-2` allows the logical connection (conjunction) of two compounds.
and-2-2	no	and, gate-2-2		
and-3	no	and, gate-3		The class `and-3` allows the logical connection (conjunction) of three compounds.
boundary	yes	gate		It is the superclass for drain and source.
drain	no	boundary		The class `drain` represents the output of a compound.
source	no	boundary		The class `source` represents an input to a compound. All compounds that have no prior activating (or inhibiting) reaction to them can be connected to a source component.

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
gate-3	yes	gate		abstract class for logical connection of 3 compounds
gate-4	yes	gate		abstract class for logical connection of 4 compounds
gate-5	yes	gate		abstract class for logical connection of 5 compounds
gate-6	yes	gate		abstract class for logical connection of 6 compounds
or	yes	gate		A logic `or`-connection can easily be implemented using `activ`, alternatively an `or`-gate can be used to allow the connection of several species to one gate, if e.g. the modeler wants to replace an `and`-gate by an `or`-gate.
or-2	no	or, gate-2		inclusive disjunction ('or'-gate) of two compounds
somehow	yes	gate		The class `somehow` act as a gate where the connection is not completely clear. It allows to model incomplete truth tables.
somehow-2	no	gate-2, somehow		
not	yes	struct-library		The class `not` is used to represent an inhibitory effect on a compound. It must be used in place before any activating reaction gate (i.e. before an `active`, `or`, `and` or `somehow` gate).

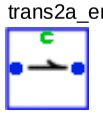
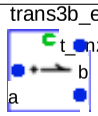
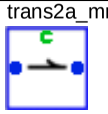
Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
struct-adapter	yes	struct-library		Adapters allow to connect a node to a terminal. The class `struct-adapter` should not be integrated into a model itself. There are two different adapter subtypes available.
struct-adapter-compound	no	struct-adapter	struct_adapter_compound	The `struct-adapter-compound` should be used to connect a compound terminal to the outer module.
struct-adapter-gate	no	struct-adapter	struct_adapter_gate	The `struct-adapter-gate` should be used to connect gates to the surroundings.
struct-model	yes	module		The class `struct-model` should be used as a superclass for all logical models and modules which are using the elements of the struct-library, It serves as marker for logical models where specialized functionality is loaded.
terminal	yes	stuctural-modeling-entity		
term-struct	yes	terminal		
term-struct-compound	no	term-struct	term_struct_compound	
term-struct-gate	yes	term-struct		
term-struct-input	no	term-struct-gate	term_struct_input	
term-struct-output	no	term-struct-gate	term_struct_output	

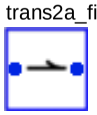
E The basic elements of the library for dynamic modeling

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
module	yes	structural-modeling-entity		Superclass containing all user-defined classes. This class is internal to ProMoT and cannot be modified or removed.
dynamic-model	yes	module		Superclass for DAE models in ProMoT.
library	yes	module		Superclass for all library elements.
storage	yes	library		Superclass for species in the network.
conc-sink	no	help, storage	conc_sink	Sink of concentration (signal). Can be used for instance to close a module containing a signal output (`term-concentration-out`).
storage-intra	no	storage	storage_intra	Intracellular storage with flux-terminal. Contains a differential equation. parameter: *c*0 (Initial concentration)
storage-intra-const_c	no	storage	storage_intra_const_c	Constant concentration with concentration (signal)-terminal. parameter: *c*0 (Constant concentration)
storage-intra_s	no	storage	storage_intra_s	Constant total concentration with terminal `term-storage-stat`. Computes a concentration given the species total concentration as a parameter (*c*0) and the concentration of its other forms given as an input. parameter: *c*0 (Total constant concentration).
storage-source	no	help, storage	storage_source	Constant concentration with flux-terminals. Deprecated class: will be removed in next version of the library because it can be replaced by `storage-intra-const_c` with an adapter.

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
reaction	yes	library		Superclass containing all reaction definitions.
kinetics	yes	reaction		Superclass containing all kinetics.
catalytic	yes	kinetics		Superclass containing all catalytic kinetics.
en1	yes	catalytic, enz-0eff		Enzymatic reaction with first-order kinetics. rate: $r = sa \cdot enzyme \cdot species_in$ parameter: sa.
mm	yes	catalytic, enz-0eff		Irreversible Michaelis-Menten kinetics with enzyme rate: $r = \frac{sa \cdot enzyme \cdot species_in}{ks1 + species_in}$ parameters: sa, $ks1$.
formal	yes	kinetics		Superclass describing the reaction types.
irreversible	yes	formal		Superclass containing all irreversible reactions.
fi1	yes	irreversible		A $\longrightarrow$ B with irreversible first-order kinetics rate: $r = k1 \cdot species_in$ parameters: $k1$.
channel2a-fi1	yes	fi1		Deprecated class: will be removed in next version.
fi2a	yes	irreversible		2 A $\longrightarrow$ B with irreversible Rate Law kinetics. rate: $r = k1 \cdot species_in^2$ parameter: $k1$.
reversible	yes	formal		Superclass containing all reversible reactions.
fr1	yes	reversible		A $\longleftrightarrow$ B with reversible first-order kinetics. rate: $r = k1 \cdot species_in - k2 \cdot species_out$ parameter: $k1$, $k2$.
fr1-x	yes	reversible		Deprecated class: use fr1b instead.
fr1b	yes	reversible		A $\longleftrightarrow$ 2 B with reversible Rate Law kinetics. rate: $r = k1 \cdot species_in - k2 \cdot species_out^2$ parameter: $k1$, $k2$.
fr2	yes	reversible		A + B $\longleftrightarrow$ C with reversible first-order kinetics. rate: $r = k1 \cdot species_in1 \cdot species_in2 - k2 \cdot species_out$ parameter: $k1$, $k2$.
fr2a	yes	reversible		2 A $\longleftrightarrow$ B with reversible Rate Law kinetics. rate: $r = k1 \cdot species_in^2 - k2 \cdot species_out$ parameter: $k1$, $k2$.
fr4	yes	reversible		A + B $\longleftrightarrow$ C + D with reversible first-order kinetics. rate: $r = k1(\cdot species_in1 \cdot species_in2 - v \cdot species_out1 \cdot species_out2)$ parameter: $k1$, v (equilibrium constant).
channel4a-fr4	yes	fr4		Deprecated class: will be removed in next version.

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
structure	yes	reaction		Superclass containing all possible reaction structures.
enz-0eff	yes	structure		Defines an input terminal for an enzyme (signal). Adds a `term-concentration-in`.
s--b	yes	structure		$\emptyset \longrightarrow B$ Defines a reaction structure with one output and no input.
s-2a-b	yes	structure		$2\,A \longrightarrow B$ or $2\,A \longleftrightarrow B$ Defines a reaction structure with one output and one input with a stoichiometric coefficient of 2.
s-2a-bc	yes	structure		$2\,A \longrightarrow B + C$ or $2\,A \longleftrightarrow B + C$ Defines a reaction structure with two outputs and one input with a stoichiometric coefficient of 2.
s-a-	yes	structure		$A \longrightarrow \emptyset$ Defines a reaction structure with one input and no output.
s-a-2b	yes	structure		$A \longrightarrow 2\,B$ or $A \longleftrightarrow 2\,B$ Defines a reaction structure with one input and one output with a stoichiometric coefficient of 2.
s-a-b	yes	structure		$A \longrightarrow B$ or $A \longleftrightarrow B$ Defines a reaction structure with one input and one output with a stoichiometric coefficient of 2.
s-a-bc	yes	structure		$A \longrightarrow B + C$ or $A \longleftrightarrow B + C$ Defines a reaction structure with two outputs and one input.
s-a-bcd	yes	structure		$A \longrightarrow B + C + D$ or $A \longleftrightarrow B + C + D$ Defines a reaction structure with three outputs and one input.
s-ab-2c	yes	structure		$A + B \longrightarrow 2\,C$ or $A + B \longleftrightarrow 2\,C$ Defines a reaction structure with two inputs and one output with a stoichiometric coefficient of 2.
s-ab-c	yes	structure		$A + B \longrightarrow C$ or $A + B \longleftrightarrow C$ Defines a reaction structure with two inputs and one output.
s-ab-cd	yes	structure		$A + B \longrightarrow C + D$ or $A + B \longleftrightarrow C + D$ Defines a reaction structure with two inputs and two outputs.

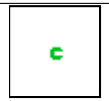
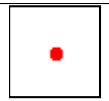
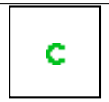
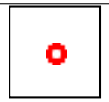
Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
trans2a-en1_r	no	en1, s-a-b_r		A $\xrightarrow{+E}$ B with irreversible first-order kinetics. $\frac{dB}{dt} = -\frac{dA}{dt} = sa \cdot E \cdot A$
trans3b-en1_r	no	en1, s-a-bc_r		A $\xrightarrow{+E}$ B, A $\xrightarrow{+E}$ C with irreversible first-order kinetics. $\frac{dB}{dt} = \frac{dC}{dt} = -\frac{dA}{dt} = sa \cdot E \cdot A$
trans2a-mm_r	no	mm, s-a-b_r		A $\xrightarrow{+E}$ B with irreversible Michaelis-Menten kinetics. $\frac{dB}{dt} = -\frac{dA}{dt} = \frac{sa \cdot E \cdot A}{ks1+A}$
trans2a-1o_r	no	trans2a-mm_r		Deprecated class: Use trans2a-en1_r instead.
trans2b-mm_l	no	mm, s-a-2b_l		A $\xrightarrow{+E}$ 2 B with irreversible Michaelis-Menten kinetics. $\frac{1}{2}\frac{dB}{dt} = -\frac{dA}{dt} = \frac{sa \cdot E \cdot A}{ks1+A}$
trans1a-fi1_l	no	fi1, s-a-_l		A $\longrightarrow \emptyset$ with irreversible first-order kinetics. $\frac{dA}{dt} = -k1 \cdot A$ Used for protein degradation.

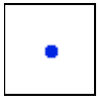
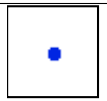
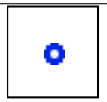
Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
trans2a-fi1_r	no	fi1, s-a-b_r		A $\longrightarrow$ B with irreversible first-order kinetics. $\frac{dB}{dt} = -\frac{dA}{dt} = k1 \cdot A$
trans2c-fi2a_r	no	fi2a s-a-b_r		2 A $\longrightarrow$ B with irreversible Rate Law kinetics. $\frac{dB}{dt} = -\frac{1}{2}\frac{dA}{dt} = k1 \cdot A^2$.
trans2a-fr1_r	no	fr1, s-a-b_r		A $\longleftrightarrow$ B with reversible first-order kinetics. $\frac{dB}{dt} = -\frac{dA}{dt} = k1 \cdot A - k2 \cdot B$.
trans2x-fr1-x_r	no	fr1-x, s-a-2b_r		Deprecated class: use trans2b-fr1_l instead.
trans2b-fr1_l	no	fr1b, s-a-2b_l		A $\longleftrightarrow$ 2 B with reversible Rate Law kinetics. $\frac{1}{2}\frac{dB}{dt} = -\frac{dA}{dt} = k1 \cdot A - k2 \cdot B^2$.
trans3a-fr2_r	no	fr2, s-ab-c_r		A + B $\longleftrightarrow$ C with reversible first-order kinetics. $\frac{dC}{dt} = -\frac{dA}{dt} = -\frac{dB}{dt} = k1 \cdot A \cdot B - k2 \cdot C$.

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
trans2c-fr2a_r	no	fr2a, s-2a-b_r	trans2c_fr2a_r	2 A $\longleftrightarrow$ B with reversible Rate Law kinetics. $\frac{dB}{dt} = -\frac{1}{2}\frac{dA}{dt} = k1 \cdot A^2 - k2 \cdot B.$
trans4a-fr4_d	no	fr4, s-ab-cd_d	trans4a_fr4_d	A + B $\longleftrightarrow$ C + D with reversible first-order kinetics. $\frac{dC}{dt} = \frac{dD}{dt} = -\frac{dA}{dt} = -\frac{dA}{dt} = k1(\cdot A \cdot B - v \cdot C \cdot D)$
special-reaction	yes	reaction		Superclass containing specific reactions.
constant-rate-conc	no	help, special-reaction	constant_rate_conc	Deprecated class: will be removed in next version of the library.
constant-rate-flux	no	help, special-reaction	constant_rate_flux	$\emptyset \longrightarrow$ B.Constant Rate with flux-terminals. rate: $r = rate$ parameter: $rate$. $\frac{dB}{dt} = rate$ Used for instance for a constitutive protein synthesis.

Class name	***Abstract***	***Superclasses***	***Icon***	***Description***
help	yes	library		Superclass containing "accessories" for making basic operations on signals or converting terminals.
sum	yes	help		These classes are needed for adding or multiplying signals, eg. enz_tot=enz1+enz2
prod-2-signal	no	sum		Product of 2 concentrations. $output = k1 \cdot input1 \cdot input2$ parameter: $k1$.
div-2-signal	no	prod-2-signal		Product of 2 signals with one having a positive effect and one having a negative effect. $output = \frac{input1}{1 + k1 \cdot input2}$ parameter: $k1$.
prod-3-signal	no	prod-2-signal		Product of 3 concentrations. $output = k1 \cdot input1 \cdot input2 \cdot input3$ parameter: $k1$.
prodsum-2-signal	no	sum		Sums and multiplies 2 concentrations, for e.g. test effect of one of them and/or product of both. $output = k1 \cdot input1 + k2 \cdot input2 + kprod \cdot input1 \cdot input2$ parameters: $k1$, $k2$, $kprod$.
sum-2-signal	no	sum		Sum of 2 concentrations. $output = k1 \cdot input1 + k2 \cdot input2$ parameters: $k1$, $k2$.
res-2-signal	no	sum-2-signal		Difference between 2 signals (like sum-2-signal but with a negative sign). $output = k1 \cdot input1 - k2 \cdot input2$ parameters: $k1$, $k2$
sum-3-signal	no	sum-2-signal		Sum of 3 concentrations. $output = k1 \cdot input1 + k2 \cdot input2 + k3 \cdot input3$ parameters: $k1$, $k2$, $k3$.

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
sum-4-signal	no	sum-3-signal	sum_4_signal	Sum of 4 concentrations. $output = k1 \cdot input1 + k2 \cdot input2 + k3 \cdot input3 + k4 \cdot input4$ parameters: $k1$, $k2$, $k3$, $k4$.
sum-2-stat	no	sum	sum_2_stat	Splits a stat-terminal into 2. $output1 = output2 = input$
sum-3-stat	no	sum-2-stat	sum_3_stat	Splits a stat-terminal into 3. $output1 = output2 = output3 = input$
sum-4-stat	no	sum-3-stat	sum_4_stat	Splits a stat-terminal into 4. $output1 = output2 = output3 = output4 = input$
term-conv	yes	help		Superclass containing elements for converting different terminal types.
cf2co	no	term-conv	cf2co	Flux → Signal Converts a `cf-term` to a `co-term`.
cf2co-vis	no	cf2co	cf2co_vis	Deprecated class: Use cf2co instead
cs2cf	no	term-conv	cs2cf	Stat → Flux Converts a `cs-term` to a `cf-term`.
cs2cf-mod	no	term-conv	cs2cf_mod	Stat → Flux Converts a `cs-term` to a `cf-term` and setting the *cbound* variable of `cs-term` to the parameter *cbound_given*. parameter: *cbound_given*.
cs2co-mod	no	term-conv	cs2co_mod	Stat → Signal Converts a `cs-term` to a `co-term` and setting the *cbound* variable of `cs-term` to the parameter *cbound_given*. parameter: *cbound_given*.
dyn-adapter	yes	library		Superclass containing adapters. Useful for instance for propagating terminals already connected to other elements.
adapter-conc	no	dyn-adapter	adapter_conc	Signal → Signal
adapter-flux	no	dyn-adapter	adapter_flux	Flux → Flux

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
terminal	yes	structural-modeling-entity		Superclass containing all terminal types used to connect elements/modules.
term-concentration-in		terminal	 term_concentration_in	Input terminal for a concentration(signal).
term-reaction-stat		term-concentration-in	 term_reaction_stat	Output terminal for a concentration(stat).
term-concentration-out		terminal	 term_concentration_out	Output terminal for a concentration(signal).
term-storage-stat		term-concentration-out	 term_storage_stat	Input terminal for a concentration(stat).

Class name	*Abstract*	*Superclasses*	*Icon*	*Description*
term-reaction-flux		terminal	 term_reaction_flux	Terminal for a flux in a reaction.
term-ext-reaction-flux		term-reaction-flux	 term_ext_reaction_flux	Deprecated class: will be removed in next version of the library.
term-storage-flux		terminal	 term_storage_flux	Terminal for a flux in a storage.

F Shortcuts and modifiers of the ProMoT Visual Explorer

Operation	*Shortcut*	*Description*
Menus		
Open node menu	Click right button on entity	Opens the context menu for nodes
Open edge menu	Click right button on entity	Opens the context menu for edges
Mode: Interactive Zoom		
Zoom in	Hold down left button + Drag right	Zoom in the network
Zoom out	Hold down left button + Drag left	Zoom out the network
Pan	Hold down left button + Ctrl + Drag	Move (pan) the entire canvas
Mode: Hierarchy Zoom		
Zoom in	Click left button on entity	Zoom in to focused entity in hierarchical steps
Zoom out	Click left button on entity	Zoom out to focused entity in hierarchical steps
Fit network on screen	Click left button + Shift	Zoom out the entire network to fit on screen
Pan	Hold down left button + Ctrl + Drag	Move (pan) the entire canvas
Mode: FishEye Zoom		
Zoom in	Click left button on entity	Enlarge focused entity, shrink all other entities
Zoom out	Click left button on entity + Shift	Shrink focused entity, enlarge all other entities
Pan	Hold down left button + Ctrl + Drag	Move (pan) the entire canvas
Mode: Focus Zoom		
Zoom to	Click left button on entity	Minimal zoom to focused entity
Pan	Hold down left button + Ctrl + Drag	Move (pan) the entire canvas
Mode: Detail+Overview		
Pan	Click left button on gray box + Drag	Move (pan) the entire canvas
~~Zoom~~	~~Click left button on handle of gray box + Drag~~	~~Zoom in or out the network~~
Mode: Node Editing		
Move	Click left button on entity + Drag	Move entity inside bounds of parent
Resize (all)	Click left button on handle + Drag	Scale entity including children and label
Resize (bounds)	Click left button on handle + Shift + Drag	Resize only bounds of entity, keep size of children and label
Pan	Hold down left button + Ctrl + Drag	Move (pan) the entire canvas
Mode: Edge Editing		
Move handle	Click left button on handle + Drag	Move edge handle
Add handle	Click right button on edge for context menu	Add edge handle
Delete handle	Click right button on edge for context menu	Remove edge handle
Delete all handles	Click right button on edge for context menu	Remove all edge handles
Pan	Hold down left button + Ctrl + Drag	Move (pan) the entire canvas
Mode: Scenario 'Interaction Distance'		
Show Int Dist	Click left button on entity	Show Interaction Distance for focused entity
Show Int Dist (multi)	Click left button on entity + Shift	Show Interaction Distance for previous and focused entity
Show Int Dist (path)	Click left button on two entities + Ctrl	Show Interaction Distance for all entities on the shortest path

Table 1: Shortcuts and modifiers used in the Visual Explorer.

References

[AFGP96] A. Artale, E. Franconi, N. Guarino, and L. Pazzi. Part-Whole Relations in Object-Centered Systems: An Overview. Data and Knowledge Engineering (DKE) Journal, 20:347–383, 1996. 6.1, 6.2

[Gil97] E.D. Gilles. Netzwerktheorie verfahrenstechnischer Prozesse. Chemie-Ingenieur-Technik, 69:1053–1065, 1997. 2

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification. Sun Microsystems, Inc., second edition, 2000. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html. 1.1

[HKK+00] M. Häfele, A. Kienle, E. Klein, A. Kremling, C. Majer, M. Mangold, A. Spieker, E. Stein, R. Waschler, and K. P. Zeyer. User Manual DIVA-3.9. MPI Magdeburg, ISR Uni Stuttgart, Magdeburg, Germany, 2000. file:///usr/local/diva/std/doc/usermanual.ps. 1.2, 7.7

[KBM+06] M. Krasnyk, K. Bondareva, O. Milokhov, K. Teplinskiy, M. Ginkel, and A. Kienle. The promot/diana simulation environment. 2006. 1

[Kö00] Reto Köhler. Beschreibung Code-Generator Version 2.0 zur Erzeugung von DIVA-Grundelementen. Institut für Systemdynamik und Regelungstechnik, Universität Stuttgart, Stuttgart, Germany, 2000. file:///usr/local/diva/std/doc/cg2.0manual.ps. 1, 4.1.1, 5.1.3, A.2

[Mar96] W. Marquardt. Modellbildung und Simulation verfahrenstechnischer Prozesse. Vorlesungsmanuskript, Lehrstuhl für Prozeßtechnik, RWTH Aachen, July 1996. 2, 4.1

[Trä98] F. Tränkle. Frame Definition Language FDL: Programmer's Guide. Internal report, Institut für Systemdynamik und Regelungstechnik, Universität Stuttgart, 1998. 3, 6.3

[tSd02] JavaTM technology Software division. JavaTM 2 SDK, Standard Edition Documentation. Sun Microsystems, Inc., 2002. http://java.sun.com/j2se/1.4/docs/index.html. 1.1

Index

CodeGenerator, **6**

`:absolute-error`, **40**
aggregation, 13, **45**, 57
  hierarchy, 13
  strong, 45
  weak, 46
array, 48

behavioral-modeling-entity, 33
BNF, **70**
browser, 51

`cd`, **16**, 16
`:class`, **16**
class-object, 18
Clos, 6
container, **18**, 45
  strong, 32, 45
  weak, 46
coupling, 29, 48

`define-equation`, 16, **42**, 42, 67
`define-module`, 16, **19**, 67
`define-terminal`, 16, **29**, 29, 67
`define-variable`, 16, **35**, 37, 67
definition-command, **16**
dialog
  model output, 51, **64**
  search class, 55
`:diva-anl-file-name`, **28**
`:diva-cg-file-name`, **28**
`:diva-data-file-name`, **28**
`:diva-fortran-files-subname`, **28**
`:diva-cg-file-name`, **28**
`:diva-latex-file-name`, **28**
`:documentation`, **16**
`:dont-optimize`, **41**

EBNF, **70**
editor
  source, 51, **63**
  visual, 51, **58**
Emacs, 63
equation, 33
`:equations`, **19**

Fdl, **18**

flux, 36, 48
  coupling, 48

`:geometry-height`, **59**
`:geometry-position`, **60**
`:geometry-side`, **60**
`:geometry-width`, **59**
`:geometry-x`, **59**
`:geometry-y`, **59**
GUI, 51

heritage view, 51

`:icon`, **59**
`include`, **16**, 16
index, 17
`:index`, **17**
`:indices`, **17**, 27
inheritance, 48, 61
internal part classes, 47
`:is-a`, 61

link, 48
`:link-color`, **60**
`:links`, **24**
`:lower`, **17**

`:maximum`, **40**
Mdl, 6, 16
`mimo-tank`, 30
`:minimum`, **40**
model
  closed, **7**
  open, **7**
model output dialog, 51
`:model-name`, **17**
modeling
  goal driven, 14
modeling-entity
  structural, 19
module, 19
  composed, 19, 24
  elementary, 19
  indexed, 19, **27**
`:modules`, **24**

network-theory, 13

palette, 58
part, 45
    strong, 45, 50
    weak, 46, 50
part class, 47
parts, **18**
place, 33
`:places`, **19**
`:positive-vector-direction`, **36**, 38
potential, 36, 48
    coupling, 48
ProMoT, 6
    architecture, 7
`promot`, 51
`promot-ui`, 51
`:properties`, **17**
property attributes, 17

`:relation`, **42**
`:report`, **40**

scalar, 48
slot, 45
    weak, 45, 46
slot-path, 32, 46
source view, 51
specialization hierarchy, 14
subclass, 48
`:superclasses`, **16**
system-theoretic, 35
`:system-theoretic`, **35**, 38

`tank`, 9
terminal, 29
    elementary, 29
    indexed, 30
`:terminals`, **19**
transition, 33
`:transitions`, **19**
`two-tanks`, 24, 47

`:unit`, **40**
`:upper`, **17**

`:value`, **38**
variable, 33
    classes, 35
`:variables`, **19**, 42
vectorial direction, 35
vectorial type, 38, 48
vectorially
    directed, 36
    inwards, 36
    outwards, 36
    undirected, 36
view
    aggregation, 51, **57**
    heritage, 51, 57
    source, 51
    subclass, 51
    super-class, 51
visual editor, 51
visual explorer, 51, **61**

work pane, 58